XML Document Versioning

Shu-Yao Chien
Dept. of Computer Science
UCLA
Los Angeles, CA 90095
csy@cs.ucla.edu

Vassilis J. Tsotras
Dept. of Computer Sci. & Eng.
UC Riverside
Riverside, CA 92521
tsotras@cs.ucr.edu

Carlo Zaniolo
Dept of Computer Science
UCLA
Los Angeles, CA 90095
zaniolo@cs.ucla.edu

Abstract

Managing multiple versions of XML documents represents an important problem, because of many applications ranging from traditional ones, such as software configuration control, to new ones, such as link permanence of web documents. Research on managing multiversion XML documents seeks to provide efficient and robust techniques for (i) storing and retrieving, (ii) exchanging, and (iii) querying such documents. In this paper, we first show that traditional version control methods, such as RCS, and SCCS, fall short from satisfying these three requirements, and discuss alternative solutions. First, we enhance RCS with a temporal page clustering policy to achieve objective (i). Then, we discuss a reference-based versioning scheme that achieves both objectives (i) and (ii) and is also effective at supporting simple queries. The topic of supporting complex queries, including temporal ones, meshes with the burgeoning interest of database researchers in XML as a database description language, and in XML query languages. In this context, the XML versioning problems are akin to those of transaction time management for databases of objects and semistructured information. Nevertheless, the need to preserve the natural ordering of XML documents frequently requires different techniques.

1. Introduction

The management of multiple versions of XML documents finds important applications [15] and poses interesting technical challenges for database researchers. A first reason for exploring the problem is traditional application domains that rely on version management, such as software configuration and cooperative work. As these applications migrate to a web-based environment, they increasingly use XML for representing and exchanging information, often seeking standard vendor-supported tools for processing and exchanging their XML documents.

New applications domains are also emerging for XML versioning: an important and pervasive one is assuring link permanence for web documents. Any URL becoming invalid causes serious problems for all documents referring to it. The problem is particularly serious for search engines that then direct millions of users to pages that no longer exist. Replacing the old version with a new one, at the same location, does not cure the problem completely, since the new version might no longer contain the keywords used in the search. The ideal solution is a version management system supporting multiple versions of the same document, while avoiding duplicate storage of their shared segments. To assure link permanence, many professionally managed sites will rely on document versioning; furthermore, they will often support search and queries on multiversion documents. Specialty warehouses and archives that monitor and collect content from websites of interest, will also rely on versioning to preserve information, track the history of downloaded documents, and support queries on these documents and their history [7].

In the database community, there is much interest in XML as a general vehicle for data definition, and in powerful query languages for XML documents. In this context, XML versioning becomes yet another instance of the various issues pertaining to transaction-time databases [9]. Indeed, many of the techniques presented in this paper are inspired by similar concepts used in temporal databases. However, there are important differences, inasmuch as the reconstruction of complete documents, or large segments thereof, is here required. Thus the logical order of the document objects must now be preserved, whereas the order of tuples is immaterial in relational databases. Likewise, the version management techniques proposed for object oriented databases [9] and semistructured information [11] assume that the order between objects is not significant, while this is essential for reconstructing an XML document.

Many traditional document versioning systems, such as RCS [13], are *edit-based*. They use edit scripts to represent document changes and to reconstruct different versions incrementally. For instance, RCS [13] stores the most current version intact while previous versions 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. Rather than appending version differences at the end as RCS, SCCS [12] inserts editing operations in the original (source code) document and associates a pair of timestamps (version ids) with each document segment to specify its lifespan. 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 by different versions, and this lack of clustering can cost many additional page I/Os.

Moreover, neither RCS nor SCCS preserve the logical structure of the original document. This makes structure-related searches on XML documents difficult and expensive to support—reconstruction of a whole version might be needed before its component objects can be identified. An additional requirement is for the versioned document to be easily exchanged at the transport level. However, the edit scripts used in RCS represent a special object that cannot be easily accommodated at the transport level without XML extensions. Ideally, we would like the versioned document to be representable as an XML document, too.

Hence new approaches are needed that will efficiently perform (i) storage and retrieval, (ii) web exchange and, (iii) querying of XML multiversion documents. In this paper, we start by showing how an edit-based versioning scheme like RCS can be enhanced with a temporal page clustering policy to achieve better storage and retrieval. Then we discuss a reference-based versioning scheme that achieves all three requirements.

The paper is organized as follows: in the next section we discuss the improved edit-based scheme while in Section 3 we present the reference-based solution. A performance comparison between the two approaches appears in Section 4 while directions of future research are discussed in the conclusions (Section 5).

2. The Edit-Based Approach

For simplicity, assume that the document's evolution creates a linear sequence of temporally ordered versions: V_1, V_2, $\ldots$, V_j, where V_j is the current version. A new version (V_{j+1}) is established by applying a number of changes (object insertions, deletions or updates) to the current version (V_j); these changes are stored in a forward edit script. The reverse edit script instead records the changes that take us from a version to the previous one. There is complete duality between the two representations, and all techniques described here work with both. Thus in our discussion, we will use forward scripts which appeal to the intuition, since they represent the history of the evolution of the document. Such scripts could be generated directly from the edit commands of a structured editor, if one was used to revise the XML document; in most situations, however, they will obtained by applying, to the pair (V_j, V_{j+1}), a structured diff package [6].

The RCS scheme performs well when the changes from a version to the next are minimal. For instance, if only 0.1% of the document is changed between versions, reconstructing the 100^{th} version requires only 10% retrieval overhead. But if 70% of the document changes between versions, then retrieving the 100^{th} version could cost 70 times the effort of 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, an adaptable self-adjusting method is needed, that for small revisions operates as RCS, and stores only the delta changes, and in the case of a major revision, it stores a new version in its entirety. Furthermore, the method must be applied to individual pages, since revisions are normally not distributed uniformly through the document, and different stored pages experience different change rates.

2.1. Page Usefulness

The objects of successive versions are stored sequentially; the first version is stored in its entirety, while only the new objects (i.e., the deltas) are stored for the other versions. A page stored by a previous version contains several objects that have later been deleted or updated and are no longer a part of the current version: thus if read from disk, only a portion of that page is "useful" for the current version. 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, O_2 is deleted; at version V_3, O_3 is updated, and at version V_4, object O_5 is deleted.

```
                VERSION 1
----------------------------------------------
     Data Pages           UBCC Script E1
----------------------------------------------
  +-----------------+   ins(@A1,1),ins(@R1,2),
P1| A1, R1, T1, U1  |   ins(@T1,3),ins(@U1,4),
  +-----------------+   ins(@P1,5),ins(@I1,6),
P2| P1, I1, N1, M1  |   ins(@N1,7),ins(@M1,8),
  +-----------------+   ins(@Q1,9),ins(@Z1,10),
P3| Q1, Z1, B1, X1  |   ins(@B1,11),ins(@X1,12),
  +-----------------+   ins(@D1,13),ins(@H1,14),
P4| D1, H1, K1, L1  |   ins(@K1,15),ins(@L1,16).
  +-----------------+

                  VERSION 2
----------------------------------------------
    Data Pages           UBCC Script E2
----------------------------------------------
  +-----------------+   ins(@G2,5),ins(@T2,6),
P5| G2, T2, R2      |   ins(@R2,11),
  +-----------------+   del(15),del(17).

                  VERSION 3
----------------------------------------------
    Data Pages           UBCC Script E3
----------------------------------------------
  +-----------------+   del(3),del(4),
P6| Z1, B1, D1      |   del(4),del(8),
  +-----------------+   del(8),ins(@Z1,8),
                        del(8), del(9),
                        ins(@B1,9),del(10),
                        ins(@D1,10),del(11),
                        del(11).
```

Figure 1. Edit-Based $UBCC$ version files

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 for version V_3. For version V_4, O_3 is updated and its new value O'_3 will be stored in another page since P is full. Thus, P is only 55% useful for V_4.

Clearly, as new versions are created, the usefulness of existing pages *for the current version* diminish. We would like to maintain a minimum page usefulness, U_{min}, over all versions. Thus, when a page's usefulness falls below U_{min}, for the current version, all the records that are still valid in this page are copied (i.e., salvaged) to another page (while preserving their order). The value of U_{min} is set between 0 and 1 and represents the main performance parameter of our scheme. For instance, if $U_{min} = 60\%$, then page P falls below this threshold of usefulness at Version 4; at this point objects O_1, and O_4 are copied to the new page.

This scheme is similar to the "time-split" operation in temporal indexing [8] [14] [1]. Reconstructing a given version is then reduced to accessing only the useful pages for this version.

2.2. The Edit Script

In addition to its usefulness-based management of data pages, our UBCC scheme is different from RCS because it stores the edit script separately from the data pages (while RCS stores them in the same file). To simplify the discussion, our script will only contain insertions and deletions, and we represent an update by a deletion followed by an insertion.

We represent a document as an ordered list of objects, with $O\#$ denoting the position of object O in such a list. A simple extension to this list-based representation to capture the structure of the XML document will be discussed later.

A version V_j is represented by an edit script containing elements of two kinds:

- the insertion of object `Q` at position i of V_j, is represented by `ins(@Q,i)`, where `@Q` is the location (i.e. page number and offset) where the object `Q` is actually stored,
- `del(i)` denotes the deletion of the object that, without this deletion, would occupy position i in V_j.

We now introduce the storage representation for the UBCC script with the help of an example involving three successive versions of a document. Each version is externally represented by a logical edit script, produced, e.g., by a structured document editor or a diff procedure.

The script for Version 1 is simply a sequence of inserts to create the first version: *insert A1, R1, T1, U1, P1, I1, N1, M1, Q1, Z1, B1, X1, D1, H1, K1, L1*.
Then, Version 2 is created by the following changes:
insert G2,T2 after U1#, insert R2 after M1#,
delete X1#, delete K1#.
Here *delete K1#* denotes the deletion of object *K1* from the previous version. Object *K1* occupies position 15 in the list representing the previous version; thus we assume that the diff procedure that computes the changes between these two versions represents this deletion by *delete 15*. Figure 1 shows how Version 1 and Version 2 are stored as two separate files, one containing data pages and the other containing the UBCC Script. The data pages simply store the objects in the order in which they have been created. For the UBCC Script, we have that, since in Version 1 *U1# =4* , then *insert G2,T2 after U1#* is represented as *ins(G2, 5), ins(T2, 6)* in Version 2. Then, *insert R2# after M1* is translated into *ins(R2, 11)*, since *M1# =8*, in Version 1, and two inserts and no delete have taken place before this: thus *R2#* $= 8 + 2 + 1 = 11$ in Version 2. In general, we find the position for an inserted object in the current version, by taking the position of the 'after' element in the previous version, adding the number of preceding insertions, and subtracting the number of preceding deletions. Finally *delete X1#, delete K1#* is translated into: *del(15), del(17)*.

Now say that Version 3, is generated by the following changes:

delete T1#, delete G2#, delete T2#, delete R2#,
delete Q1#, delete H1#, delete L1#,

and that U_{min} has been set to 70%. Also, assume that objects are of the same size and four objects fill a page. Then, pages P3 and P4 are 75% useful for Version 2, thus no copying was needed. Now, for Version 3, pages P3, P4, and P5 become 50%, 25%, and 0% useful, respectively. Then, these three pages have fallen below the threshold of usefulness, and their valid objects—namely, *Z1, B1*, and *D1*—must be copied. New objects and copied objects are stored into a new data page P6 in their sequential order for Version 3 (thus `@Z1` in edit script, E3, for Version 3, points at `Z1` in P6— Figure 1). For each copied object, a pair of entries—one deletion followed by one insertion—are added to the edit script.

The policy of copying long-lived objects to new pages, assures that all the objects valid for a given version are clustered closely together.

2.3. Version Reconstruction.

Consider retrieving version 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 edit scripts. 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 edit script, E3. The first entry is *del(3)*: thus, the first two objects are missing 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 entry in E2, *ins(G2, 5)*. We get a gap again and need to retrieve two objects from its 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 for output. The data page P1 remains in main memory, where from the next valid object for Version 3, i.e., *U1*, is retrieved. The reconstruction of Version 3 resumes with *del(3)*. Therefore, Version 3 requests the next object of Version 2; to answer the request, Version 2 needs to retrieve its third object (because its first two objects have been retrieved in the previous request). But, since the current entry for Version 2 is *ins(G2,5)* it must request this object from Version 1. Version 1 responds to the request of Version 2 by returning record *ins(T1,3)*, which is then returned to Version 3, where is nullified by the delete operation *del(3)*. At this point the third object of Version 3 has not been retrieved yet. So another next-object request is issued from Version 3 to Version 2 and, recursively, to Version 1, thus Version 3 is returned record *ins(U1, 4)*, i.e., the third object of Version 3. This gap-filling procedure continues through the script E3 until all objects of Version 3 are retrieved. A detailed description of version reconstruction algorithm was given in [2].

2.4 Performance

An experimental study of the performance the UBCC scheme is presented in Section 4, where the scheme is compared to (i) RCS, (ii) storing complete versions, and (iii) the reference based scheme discussed in the next section. The results of this comparison show that UBCC performs well, often achieving both the better storage performance of (i) and the better retrieval performance of (ii). Another study presented in [3] shows that this scheme performs better than techniques such as multi-version B-trees [1] and partially persistent lists that have been used in transaction-time databases and persistent storage managers [8, 14].

The UBCC scheme also provides better support for queries, since many searches can be performed directly on the script, rather than the data pages. The overall length of the script can be kept to a small percentage of the overall size of the data by taking regular snapshots of the same. The details of this improvement are given in [3], where we also prove that the overall storage space used remains linear in the number of changes in the document's version history.

The average retrieval efficiency grows with U_{min}. Assuming a buffer of i pages in memory, to reconstruct version V_i we need to read edit scripts $E_i \cdots E_1$ and the pages that are useful for version V_i. If B denotes the page size, the number of useful pages of version V_i is bounded by $\frac{1}{U_{min}} \times size(V_i)/B$.

2.5. Discussion

Several techniques can be used to support the reconstruction of a structured XML document from the linear list of its objects. One solution consists in including more information in the script record denoting the insertion of an object N: for instance, we can include $G\#$, where G is the parent of N.

A second solution, consists in allowing entries such as $ins(`[', N\#)$ to denote that a left bracket must be inserted before N. Thus, by using left and right brackets, we can represent the structure of the document. Transformations on the structure of the document are then specified by deletions and insertions of bracket elements. Bracket elements are only stored in the script and not in the data pages. A third solution consists of replacing $N\#$ (which basically corresponds to the preorder traversal number of node N in the document tree)

```
            VERSION 1
---------------------------------------
   Snapshot        Reference-Based
---------------------------------------
A1, R1, T1, U1,     A1, R1, T1, U1,
P1, I1, N1, M1,     P1, I1, N1, M1,
Q1, Z1, B1, X1,     Q1, Z1, B1, X1,
D1, H1, K1, L1,     D1, H1, K1, L1.

            VERSION 2
---------------------------------------
    Snapshot       Reference-Based
---------------------------------------
A1, R1, T1, U1,     (V1, (1, 4), 1), G1,
G2, T2, P1, I1,     T2, (V1, (5, 8), 7),
N1, M1, R2, Q1,     R2, (V1,(9,11),12),
Z1, B1, D1, H1,     (V1, (13, 14), 15),
L1.                 (V1, (16, 16), 17).

             VERSION 3
---------------------------------------
    Snapshot       Reference-Based
---------------------------------------
A1, R1, U1, P1,     (V2, (1, 2), 1),
I1, N1, M1, Z1,     (V2, (4, 4), 3),
B1, M3, D1, H1,     (V2, (7, 10), 4),
L1.                 (V2, (13, 14), 8),
                    M3, (V2,(15,17),11).
```

Figure 2. The reference-based scheme

with the full tree address of N. This representation is discussed in [5].

While UBCC provides several improvements with respect to RCS, its generality and flexibility remain limited, and it is not suitable as an external representation for exchanging documents. These problems are addressed by the *reference-based versioning scheme* discussed next.

3. The Reference-Based Scheme

While edit-based approaches focus on representing changes, our new scheme concentrates on representing the parts that have remained unchanged, i.e., the *common segments* between two successive versions.

Versions in the reference-based scheme are represented as a *list* of the following two kinds of objects:

- *reference records* which denote maximum common segments shared between the new version and the previous version, and
- *actual object records*.

Let us use the same example as in the previous section. Then, the reference-based representation of the initial version, Version 1, is the version itself. Then, Version 2, is represented by the new objects inserted in Version 2 and the five maximal common segments shared with Version 1, as follows:

```
(A1,R1,T1,U1), G2, T2, (P1,I1,N1,M1),
R2, (Q1,Z1,B1), (D1, H1), (L1)
```

where the common segments are shown in parentheses. In our storage representation each common segment is represented by a *reference record* of the form:

(V#, Common_Segment_Reference, New_Position)

where *V#* denotes the previous version, and *Common_Segment_Reference* is a pair of position values specifying the starting position and end position of the common segment in the previous version *V#*, and *New_Position* denotes the position of the common segment in the new version. For example, the reference record *(V1,(1,4),1)* refers to the first common segment, `(A1,R1,T1,U1)`, which starts from the first object of Version 1 and ends at the fourth object. The position value, 1, implies that this segment is placed at the beginning (first position) in the new version. Therefore, Version 2 is represented as the following list:

(V1,(1,4),1),G2,T2,(V1,(5,8),7),R2,
(V1,(9,11),12),(V1,(13,14),15),(V1,(16,16),17)

Similarly, Version 3, is generated from Version 2 by deleting *T1, (G2, T2), (R2, Q1)* and adding *M3* after *B1*. Thus, Version 3 is represented as follows:

(V2,(1,2),1),(V2,(4,4),3),(V2,(7,10),4),
(V2, (13, 14), 8), M3, (V2, (15, 17), 11).

This reference-based representation can be constructed directly from the edit script, and vice versa.

Restructuring and Duplicating. It is often the case that two sections of the old version are switched in a new version. Also some passages and footnotes might be repeated at various points in the documents. The reference-based representation handles these changes via simple reference records, whereas the edit-based scheme requires the re-insertion of the repeated objects.

3.1. Version Retrieval

When reconstructing a version from the reference-based representation, some of the version objects are materialized by traversing reference records. Let's take Version 3 as an example. The first reference record of Version 3, `(V2,(1,2),1)`, refers to the first two objects of Version 2. To locate the first two objects of Version 2 its reference-based representation is checked. The first reference record of Version 2, `(V1,(1,4),1)`, implies that its first two objects are the first two objects of Version 1. Therefore, recursively, the first two objects of Version 3, namely, *A1* and *R1*, are found in Version 1. The second record of Version 3, `(V2, (4, 4), 3)`, refers to the fourth object of Version 2. Again, the first reference record of Version 2 is used to refer to the fourth object of Version 1 where the actual object, U1, is found. The above recursive segment locating procedure is applied to each reference record until corresponding actual object segments are found.

```
                  VERSION 1
-------------------------------------------
          Data Pages            Page Index
-------------------------   ----------------
 P1:  A1   R1   T1   U1         P1(1,4)
 P2:  P1   I1   N1   M1         P2(5,8)
 P3:  Q1   Z1   B1   X1         P3(9,12)
 P4:  D1   H1   K1   L1         P4(13,16)

                  VERSION 2
-------------------------------------------
           Data Pages         Page Index
-------------------------------------------
 P5:  (V1, (1, 4), 1),          P5(1,10)
      G2(5), T2(6),             P6(11,17)
      (V1, (5, 8), 7)
 P6:  R2(11), (V1,(9,11),12),
      (V1, (13, 14), 15),
      (V1, (16, 16), 17),

                  VERSION 3
-------------------------------------------
           Data Pages         Page Index
-------------------------   ----------------
P7:  A1(1), R1(2), U1(3),       P7(1, 4)
     P1(4)
P8:  I1(5), N1(6), M1(7),       P8(5, 8)
     Z1(8)
P9:  B1(9), M3(10),             P9(9, 13)
     (V2,(15,17),11)
```

Figure 3. The Reference-based scheme with usefulness-based clustering

3.2. Modified Page Usefulness

To match the performance of the edit-based UBCC scheme, we need to extend the *usefulness* clustering approach to the reference-based scheme. The extension is not trivial, since pages now contain both data and references. To simplify the discussion, let us assume that reference records and object records have the same size (whereas the former are normally smaller), and a page holds four records.

Consider now page P5. This holds two objects and two references. After recursively expanding these references to actual objects, we obtain a "logical" segment for Version 2 that starts from its first object and extends until its 10^{th} object. This segment will be denoted $S(P5)$. To read segment $S(P5)$ we must now access 3 pages (i.e., P1, P2 and P5). Alternatively, we can store the 10 data records of segment $S(P5)$ in 2.5 pages. Thus the overall usefulness of those three pages for Version 2 is $2.5/3 = 83\%$—or, in terms of records, out of the 12 records read 10 are useful. Then, we will say that the usefulness of page $P5$ is 83% (in reality this is the combined usefulness of $P5$ and the pages transitively referenced by it).

The next four records of Version 2 are stored in a new page, P6. The logical segment of page P6 starts from the eleventh object of Version 2 (R2) and extends until the version's last object, L1. Materializing this logical segment requires accessing pages P6, P3 and P4. Since objects X1 and K1 are deleted, only seven out of the twelve records from these three pages are useful. As a result, page P6 is 58% useful. We now define the usefulness of a page P that contains object and reference records for a version V.

Definition: Let $S(P)$ the document segment obtained by materializing page P, and let n be the number of pages accessed during the materialization. Then, with B the size of a page, the usefulness of P is:

$$\frac{size(S(P))}{B \times (n+1)}$$

□

To guarantee low I/O cost, we define a minimum required usefulness U_{min}. Before storing a new page P for the current version, we check its usefulness, and if it is below U_{min}, we store $S(P)$ instead of P. For instance, if we set $U_{min} = 40\%$ then we see that the usefulness of pages P1–P6 storing the objects of Versions 1 an 2 are above this threshold (Figure 2). But now as we start a new page, say page P' for Version 3, we see that to materialize $S(P')$ we must access P5, P1, P2, P3 and P6, where we find 9 valid objects. Thus, the usefulness of P' is 37.5%, which is below the threshold; therefore, we copy those 9 objects and store them in pages P7, P8 and P9 as shown in Figure 3.

Details of the version retrieval algorithm for the reference-based scheme are given in [4], where we also show that its worst-case storage cost and version-retrieval cost are the same as the UBCC scheme [4].

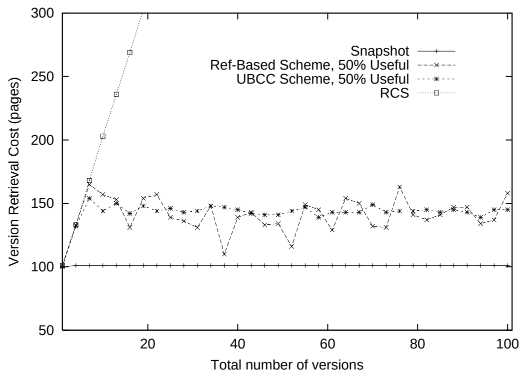
3.3. Transport Level

An improved reference scheme has been proposed in [5], where objects are referenced by their tree address in the XML document. Since such references are allowed for XML documents, it then become possible to represent the whole history of an XML document as yet another XML document. In fact, the DTD or the schema of such a history can be generated the automatically from those of the original (unversioned) document [5]. This representation is very suitable when multiversion documents need to be exchanged between different sites (transport level).

With the help of page indexes, such as those shown in Figure 3, the reference-based scheme is also conducive to efficient query processing [5].

4. Performance Analysis

We compared the performance of the reference-based scheme with the edit-based scheme and the basic RCS approach. As a baseline case we also report the performance of a "snapshot" scheme, that simply stores a



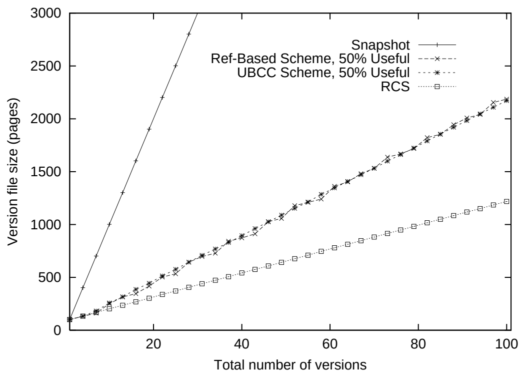
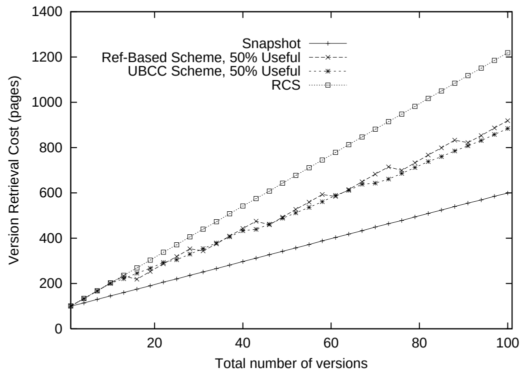


Figure 4. Version retrieval and storage cost with 50% usefulness requirement.



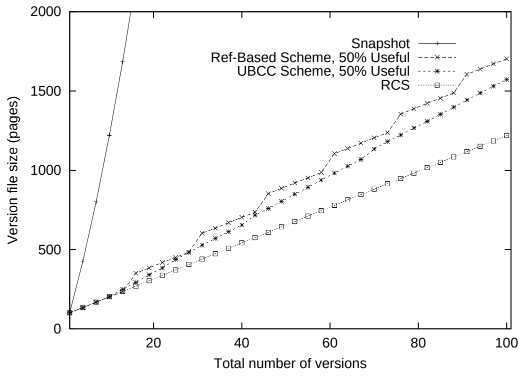


Figure 5. Version retrieval and storage cost with increasing document size.

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. In the first set of experiments, 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.

A second set of experiments evaluated how the schemes behave when the documents grow in size. In this second set, insertions add up to 10% of the document size and deletions to 5% of document size (for a 5% of net growth).

Figure 4 shows the results of the first set. Version retrieval cost is measured as the number of page I/O's needed to reconstruct a version. Obviously, the snapshot scheme has the minimum version retrieval cost, and the maximum storage cost, since each version is already stored in its entirety on disk. Symmetrically, the RCS scheme requires the least storage but has the largest average retrieval cost. The usefulness-based schemes tradeoff between these two extremes and deliver intermediate performance. The figure shows that the average retrieval cost for the usefulness-based schemes remains linear in the size of the reconstructed version by a coefficient that is controlled by U_{min}. In the first experiment, the average version size was kept unchanged to about 100 pages. The retrieval cost of the edit-based scheme is approximately parallel to the horizontal axis, for a total cost near 150 pages, for $U_{min} = 50\%$.

In the reference-based scheme when the usefulness of a segment falls below the threshold, several new pages are normally generated. Because of this larger granularity, retrieval for the reference-based scheme shows more substantial fluctuations around the smoother line of the edit-based scheme. Thus, some valleys in the reference-based curve approach the performance of the snapshot case; its peaks exceed the 150 page level but remain well below the theoretical worst case of 200 pages $U_{min} = 50\%$. On the average, our experiments clearly show that the retrieval and storage performance of reference-based and edit-based schemes are very close to each other, for the same usefulness factor.

Figure 5 results for the second set of data that describes documents growing in size. The version retrieval cost for the reference-based scheme fluctuates around the edit-based scheme, while their storage cost remain close to each other. Similar results were obtained for documents which are shrinking in size [4].

5. Conclusions

Versioning schemes for XML documents can play an important role in the management of web based information. However, traditional techniques such as RCS and SCCS are not up to the task and there is a need for new and improved techniques that achieve better performance at the physical level and the logical level.

For the physical level, we have proposed a temporal clustering technique based on page usefulness to trade off storage efficiency with retrieval efficiency and optimize the overall performance. By combining this technique with an edit-based representation, we have derived the UBCC scheme, which improves on RCS by reducing the average version retrieval cost, at the price of a small storage overhead. By using a separate edit script this scheme also assures a faster retrieval of selected objects.

Then, we introduced the reference-based scheme which preserves the basic structure of the document, by identifying the objects shared with the previous version. At the logical level, this scheme has better properties than the edit-based UBCC scheme; in fact, by using tree nodes address, a multiversion XML document can be represented as a standard XML document whose schema or DTD can be constructed automatically from those of the original (unversioned) document [5]. This representation is very suitable when multiversion documents need to be exchanged between different sites (transport level). Also this representation is conducive to efficient query processing [5].

Querying XML document represents a research area of growing interest. We expect that this trend will also impact versioning techniques, and present problems similar to those encountered in transaction-time temporal databases. In particular, efficient support will be needed for traditional (snapshot) queries, temporal queries on the evolution of the document, and various combinations of the two. Techniques for supporting complex queries on XML databases often decompose the document and rely on various node-numbering schemes to preserve its structure [10]; these schemes can be extended to preserve the node-numbers of unchanged objects when the document is updated [10]. We conjecture that, by using such node numbering schemes, the effectiveness of time-stamping methods of temporal databases can be extended to XML document versioning, and we plan to investigate this approach in the future. In general, problems such as, (i) generalizing XML query languages with constructs for expressing queries on document evolution, and (ii) ensuring efficient support for temporal and nontemporal queries under different versioning schemes, provide interesting research challenges for database technology and web-based information systems of the future.

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] S-Y. Chien, V.J. Tsotras, and C. Zaniolo, *"Version Management of XML Documents"*, WebDB 2000 Workshop, Dallas, TX, 2000, pp 75-80.

[3] S-Y. Chien, V.J. Tsotras, and C. Zaniolo, *"A Comparative Study of Version Management Schemes for XML Documents"*, TimeCenter Technical Report TR-51, Sep. 2000.

[4] S.-Y. Chien, V.J. Tsotras, and C. Zaniolo, *"Copy-Based versus Edit-Base Version Management Schemes for Structured Documents,"* In Proc. 11-th RIDE Workhshop, Heidelberg, Germany, April, 2001.

[5] S.-Y. Chien, V.J. Tsotras, and C. Zaniolo, *"Efficient Management of Multiversion Documents by Object Referencing"*, In Proc. VLDB 2001, Roma, Italy, Sept., 2001.

[6] G. Cobena, S. Abiteboul, A. Marian, *"XyDiff Tools Detecting changes in XML Documents"*. http://www-rocq.inria.fr/ cobena.

[7] A. Marian, et al., *Change-centric management of versions in an XML warehouse*. In Proc. VLDB 2001, Roma, Italy, Sept., 2001.

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

[9] G. Ozsoyoglu and R.T. Snodgrass, *Temporal and Real-Time Databases: a Survey*, IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No.4, pp. 513-532, 1995.

[10] Q. Li and B. Moon, *"Indexing and querying XML data for regular path expressions"*, In Proc. of VLDB 2001, Roma, Italy, September, 2001.

[11] S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom, *"Change Detection in Hierarchically Structured Information"*, In Proc. ACM SIGMOD, 1996.

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

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

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

[15] webdav, WWW Distributed Authoring and Versioning *www.ietf.org/html.charters/webdav-charter.html*