Copy-Based versus Edit-Based Version Management Schemes for Structured Documents

Shu-Yao Chien*† Vassilis J. Tsotras‡ Carlo Zaniolo§

Abstract

Managing multiple versions of XML documents and semistructured data represents a problem of growing interest. Traditional version control methods, such as RCS, use edit scripts representing changes in the document to support the incremental reconstruction of different versions. The edit-based approaches have been recently enhanced with a replication scheme called UBCC [2]. UBCC is based on the notion of page usefulness *and ensures effective management for multi-version documents in terms of both retrieval and storage cost. These improvements notwithstanding, the edit-based representation suffers from limited generality and flexibility–e.g., it cannot represent changes such as rearranging the document or duplicating parts of its content. To solve these problems, the paper proposes a copy-based UBCC versioning scheme, which also provides a simpler format for the electronic exchange of multi-version documents. With the objective of matching the performance of the edit-based UBCC technique, we develop algorithms that enhance the copy-based UBCC scheme with page usefulness management. We also present results of various experiments that test the storage and retrieval performance of the new copy-based approach, and compare it with that of the edit-based UBCC approach.*

1. Introduction

The problem of managing multiple versions for XML and semistructured documents is of significant interest for content providers and cooperative work. The XML standard is considering this problem at the transport level. The WEBDAV working group [8] is developing a standard extension to HTTP to support version control, meta data and name space management, and overwrite protection.

Traditional document version management schemes, such as RCS [6] and SCCS [5], are *line-oriented* and suffer from various limitations and performance problems. For instance, RCS [6] 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 [5] 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.

Recently, we enhanced the family of edit-based schemes with a replication scheme, called $UBCC$ [2]. To ensure that all versions can be reconstructed with I/O cost that is proportional to the version's size, the $UBCC$ scheme clusters valid objects of a given version in few data pages. This is achieved through the notion of *page-usefulness*. 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. A version is then reconstructed by only accessing *useful* pages (i.e., pages filled mostly with objects that are valid

*Computer Science Department, UCLA. Email:csy@cs.ucla.edu.

†This work has been supported by NSF under grants IIS-0070135 and IIS-9907477 and by the Department of Defence.

‡Department of Computer Science and Engineering, UC Riverside. Email:tsotras@cs.ucr.edu.

§Computer Science Department, UCLA. Email:zaniolo@cs.ucla.edu.

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.

However, in spite of these improvements, the edit-based representation of versions suffers from limited generality and flexibility. For example, it can not efficiently represent changes such as document content rearranging and document restructuring. To solve these problems, we propose a new *copy-based versioning scheme*, which gets rid of edit scripts and uses the concept of *common segments* to represent versions. This new scheme also provides a simpler, more flexible format which can be used for the electronic exchange of multi-version documents, WWW-based cooperative authoring and versioning activities. In addition, with the objective of matching the $UBCC$'s performance, we develop algorithms that enhance the copy-based scheme with the usefulness-base page management method used in $UBCC$. After formalizing the algorithms used in the two methods, we present the results of various experiments to test and compare the performance of these two strategies.

The rest of the paper is organized as follows: in the next section, we review the edit-based $UBCC$ scheme. Then, the new copy-based scheme and its usefulness-based improvement are proposed in Section 3 and 4. The performance is evaluated in Section 5.

2. The Edit-Based $UBCC$ Scheme

The RCS scheme performs well 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, if 70% of the document changes between versions, then 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, an adaptable self-adjusting method is needed, that in one 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. A detailed study of the performance of this method was presented in [3], where its performance was compared against techniques that use multi-version B-trees and partially persistent lists. In general, the performance of the new usefulness-based scheme was better than that of the other schemes, which are based on techniques originally developed for transaction-time databases and persistent object stores [3].

2.1. 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. To further simplify the discussion, we only consider insertion and deletion changes (an update can be represented by a deletion followed by an insertion of the updated object). Note that an object deletion at version V_i is not physical but logical: the previous version of the object is maintained since it is valid for some earlier version(s). Clearly, based on the changes in the version evolution, an object is associated with a *lifetime* interval of the form: [insertion_version, deletion_version). An object is valid for all versions in its 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, U_{min}, 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 [7] [4]. Reconstructing a given version is then reduced to accessing only the useful pages for this version; this is

```
                         VERSION 1
-----------------------------------------------------
      Data Pages                 UBCC Script E3
-----------------------    --------------------------
   +------------------+    ins(A1,1,p1),ins(R1,2,p1),
p1 | A1, R1, T1, U1   |    ins(T1,3,p1),ins(U1,4,p1),
   +------------------+    ins(P1,5,p2),ins(I1,6,p2),
p2 | P1, I1, N1, M1   |    ins(N1,7,p2),ins(M1,8,p2),
   +------------------+    ins(Q1,9,p3),ins(Z1,10,p3),
p3 | Q1, Z1, B1, X1   |    ins(B1,11,p3),ins(X1,12,p3),
   +------------------+    ins(D1,13,p4),ins(H1,14,p4),
p4 | D1, H1, K1, L1   |    ins(K1,15,p4),ins(L1,15,p4).
   +------------------+

                         VERSION 2
-----------------------------------------------------
      Data Pages                 UBCC Script E3
-----------------------    --------------------------
   +------------------+    ins(G2,5,p5),ins(T2,6,p5).
p5 | G2, T2, R2       |    ins(R2,11,p5),
   +------------------+    del(X1,15),del(K1,17).

                         VERSION 3
-----------------------------------------------------
      Data Pages                 UBCC Script E3
-----------------------    --------------------------
   +------------------+    del(T1,3),del(G2,4),
p6 | Z1, B1, D1       |    del(T2,4),del(R2,8),
   +------------------+    del(Z1,8),ins(Z1,8,p6),
                           del(Q1,8),del(B1,9),
                           ins(B1,9,p6),del(D1,10),
                           ins(D1,10, p6),del(H1,11),
                           del(L1,11).
```

Figure 1. Edit-Based $UBCC$ version file.

very fast since each useful page contains a good part of the requested version. Furthermore, 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 [2].

2.2. An Example

Consider the versions shown in Figure 2. In the edit-based $UBCC$ scheme, the first version is stored sequentially in pages P1, P2, P3 and P4. We have assumed that each page can contain at most 4 objects. Let U_{min} be 70%. Pages P1, P2, P3 and P4 are all 100% useful for Version 1.

Version 2 is created by the following changes: *insert G2 after U1, insert T2 after G2, insert R2 after M1, delete X1, delete K1*. After applying these changes, pages P3 and P4 become 75% useful (since X1 and K1 are not part of Version 2). Pages P1 and P2 remain 100% useful. Since all pages are useful no copying is needed. New objects G2, T2 and R2 are then stored in a new page P5.

UBCC Edit Script. To be able to reconstruct any version, we need to record an *edit script* for each version. The script for Version 2, E2, is shown in Figure 1. E2 is derived from the original edit script as follows:

- Each copied object is treated as a delete operation followed by an insert operation.
- Attach to each operation the position of its target object in the new version.

Note that, the position associated to a deleted object is its position in the new version as if it was not deleted. For example, the position of X1 is 15 in the new version if it was not deleted, so, the delete operation *del(X1)* has a position value of 15. These position values are useful for recovering the logical object order and are further discussed below.

Version 3, is generated by the following changes: *delete T1, delete G2, delete T2, delete R2, delete Q1, delete H1, delete L1* As a result, pages P3, P4, and P5 become 50%, 25%, and 0% useful, respectively. As a result, the valid objects in P3 and P4, *Z1, B1, and D1* must be copied. New objects and copied objects are stored into a new data page P6 in their sequential order in Version 3. The edit script, E3, for Version 3 is shown in Figure 1. For each copied object, a pair of operations—one deletion followed by one insertion—are added into the edit script.

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. 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(T1,3)*, 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 of E2, *ins(G2, 5, p5)*. 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. When Version 3 receives these two records, it reads the data page P1 which contains these two objects, A1 and R1, and output them. Page P1 is kept in main memory because it still contains one valid object, U1, for Version 3. The reconstruction of Version 3 continues from the previous stop point, *del(T1, 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 T1. 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 *ins(G2,5,P5)* operation. As expected, the returned record is *ins(T1,3,P1)*. The record is recursively returned to Version 3 and nullified by the delete operation, del(T1, 3). However, at this point the third object

```
                            VERSION 1
------------------------------------------------------------
      Diff              Snapshot       Copy-Based Model
----------------- ---------------- --------------------
None               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
------------------------------------------------------------
      Diff              Snapshot       Copy-Based Model
----------------- ---------------- --------------------
INS(G2 after U1)   A1, R1, T1, U1,  (V1, (1, 4), 1), G1,
INS(T2 after G2)   G2, T2, P1, I1,  T2, (V1, (5, 8), 7),
INS(R2 after M1)   N1, M1, R2, Q1,  R2, (V1,(9,11),12),
DEL(X1)            Z1, B1, D1, H1,  (V1, (13, 14), 15),
DEL(K1)            L1.              (V1, (16, 16), 17).

                            VERSION 3
------------------------------------------------------------
      Diff              Snapshot       Copy-Based Model
----------------- ---------------- --------------------
DEL(T1)            A1, R1, U1, P1,  (V2, (1, 2), 1),
DEL(G2, T2)        I1, N1, M1, Z1,  (V2, (4, 4), 3),
DEL(R2, Q1)        B1, M3, D1, H1,  (V2, (7, 10), 4),
INS(M3 after B1)   L1.              (V2, (13, 14), 8),
                                    M3, (V2,(15,17),11).
```

Figure 2. A sample copy-based version representation.

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 and Version 3 gets back the record ins(U1, 4, P1) which refers to the third objects of Version 3. This gap-filling procedure continues through the script E3 until all objects of Version 3 are retrieved.

The detailed version reconstruction algorithm is available in [2]. Furthermore, [3] presents an improvement of the scheme that uses script snapshots to improve performance. Let S_{chg} denote the total number of changes in the version evolution, which includes insert, update, and delete operations. Then, it was shown in [3] that the total size of the edit script and the size of the database (actual and copied objects) remains linear in S_{chg}. 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 only the useful pages of version V_i. If B denotes the page size, the number of useful pages of version V_i is bounded by $\frac{1}{U_{min}} \times size(V_i)/B$.

3. The Copy-Based Scheme

We will use the same sample document and versions from the previous section (shown in Figure 1) to illustrate the copy-based scheme. Versions in the copy based scheme are represented as a *list* of the following two kinds of objects:

- *reference records* which denote maximum consecutive unchanged object segments that are in common between the new version and the old version, and
- *actual object records*.

The copy-based representation of the initial version, Version 1, is the version itself. Version 2 is generated by the following changes to Version 1 :

$$Insert\ G2,\ T2\ after\ U1;\ Insert\ R2\ after\ M1;$$
$$Delete\ X1;\ Delete K1.$$

Then, Version 2, is represented by the new objects inserted in Version 2 and the five maximal common segments shared with Version 1: (A1, R1, T1, U1), (P1, I1, N1, M1), (Q1, Z1, B1), (D1, H1), and (L1).

Each common segment is represented by a *reference record* of the form:

$$(V\#, Common\_Segment\_Reference, New\_Position)$$

where *V#* is the previous version, $Common\_\ Segment\_\ Reference$ 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 in Figure 2 is generated from Version 2 by deleting T1, (G2, T2), (R2, Q1) and adding M3 after B1. Thus, Version 3 is represented as following:

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

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. Our copy based representation handles these changes via simple reference records, whereas the edit script based version requires the re-insertion of the moved sections and the repeated objects.

3.1. Version Retrieval

When reconstructing a version from the copy-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 copy-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

```
COPY(int V_NUM, int START_POS, int END_POS) {
  RESULT_LIST = NULL;
  CUR_POS = START_POS;
  LIST_ELEMENT = the list element of Version V_NUM
    whose position is at START_POS;
  while (CUR_POS <= END_POS)
  { if (LIST_ELEMENT == an actual object) {
      append LIST_ELEMENT to RESULT_LIST;
      LIST_ELEMENT = next element from version V_NUM;}
    else if (LIST_ELEMENT == (v_n, start, end, new)
      /* a reference record */) {
      CAR_of_LIST = COPY(v_n, start, start);
      CDR_of_LIST = (v_n, start + 1, end, new + 1);
      append CAR_of_LIST to RESULT_LIST;
      if (CDR_of_LIST is not an empty list)
        LIST_ELEMENT = CDR_of_LIST;
      else
        LIST_ELEMENT = next element from version V_NUM;
    }
    CUR_POS ++;}
  return RESULT_LIST;
}
```

Figure 3. List Materialization Algorithm.

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. The detailed version retrieval algorithm is shown in Figure 3.

Document Segment Retrieval. The procedure used to retrieve the whole document can also be used to retrieve any portion of the document. For instance, to materialize the segment of version V2 from object 24 till object 32 you need to simply call the list materialization algorithm of Figure 3 by (V2, 24, 32). The usefulness-based management discussed next makes use of this property.

3.2. Modified Page Usefulness

In order to improve performance, we develop algorithms that enhance the above copy-based scheme with a *usefulness* clustering scheme. This scheme is a modification of the page usefulness management used in edit-based UBCC. To simplify the discussion, we assume that reference records have the same size as the actual object records (whereas they are normally smaller). Moreover, in our examples a page holds four (object or reference) records.

In the copy-based $UBCC$ scheme, the reference and actual object records of a version are stored in data pages sequentially by their logical order. Version 1 has only actual objects, so its objects are stored sequentially in four data pages, P1, P2, P3 and P4, as shown in Figure 4.

The representation of Version 2 contains both reference and actual records. The first four of these records are stored in a new data page P5. Consider the overall"logical" segment corresponding to page P5, which we will denote S_5.

```
                        VERSION 1
-------------------------------------------------------
           Data Pages                       Page Index
-------------------------------------   ----------------
P1:  A1(1)  R1(2)  T1(3)  U1(4)             P1(1,4)
P2:  P1(5)  I1(6)  N1(7)  M1(8)             P2(5,8)
P3:  Q1(9)  Z1(10) B1(11) X1(12)            P3(9,12)
P4:  D1(13) H1(14) K1(15) L1(16)            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, R1, U1, P1                         P7(1, 4)
P8:  I1, N1, M1, Z1                         P8(5, 8)
P9:  B1, M3, (V2, (15, 17), 11)             P9(9, 13)
```

Figure 4. A sample copy-based $UBCC$ version file.

This logical segment starts from the first object in Version 2 and extends until the 10^{th} object of Version 2. To physically materialize S_5, pages P1 and P2 must also be accessed. This is because some actual objects of S_5 are physically stored in these two pages. Out of the twelve records read (i.e., the records in pages P1, P2 and P5) ten correspond to real objects (i.e., "useful" records) and two are reference records (from page P5). Collectively, 83% of three accessed data pages, P5, P1 and P2, are *useful* for S_5. We will use the *collective usefulness* for S as the usefulness of page P5. That is, *page P5 is 83% useful.* 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.

The usefulness of a page represents the I/O efficiency of materializing the overall logical segment it contains. If the usefulness of a page is U and its logical segment is S, the total size of data pages accessed for materializing S is $size(S)/U$. The 83% usefulness of page P5 implies that the total number of records accessed for materializing S_5 is 120% of the size of S_5. That is, only few more records are accessed than what is needed. Now, let's formalize the new definition of page usefulness.

Definition: Consider page P and let S be the logical segment it contains for version V. Let pages $P_1, P_2, \cdots, P_n$ be

```
INSERT() {
  for (each element, E) {
    Insert E in the accepting page until page is full;
    if (accepting page is USEFUL) {
      Write current accepting page;
      Generate a new accepting page; }
    else if (accepting page is USELESS) {
      Materialize the segment it contains;
    }}}
```

Figure 5. Version Insertion Algorithm.

the extra pages needed to be accessed for materializing S, and let B be the size of these pages. The usefulness of page P for version V is the ratio of the size of S over the total size of pages $P, P_1, P_2, \cdots, P_n$:

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

□

To guarantee low I/O cost, we define a minimum required usefulness U_{min}. A page will be called $useful$ as long as its usefulness is greater or equal than U_{min}. The next question is what to do with pages whose usefulness is below U_{min}.

For example, assume that U_{min} has been set to 40% and Version 3 is represented as in Figure 4. Consider the first four reference records of Version 3. If these records were stored in a new page, say page P', materializing the segment in P' requires reading pages P', P5, P1, P2, P3, and P6. However, only nine out of the twenty four records read are useful; that is, P' has usefulness only 37.5%. The cost of materializing the segment of P' is high: 267%, To avoid high I/O cost, it is better to materialize the actual objects of a useless page and cluster them together in pages. This effectively creates copies of actual records.

Copy Procedure. Instead of storing page P', the list materialization algorithm in Figure 3 is used to identify the actual objects of the logical segment in P'. These objects are copied and stored sequentially in new data pages, P7, P8, and P9 (Figure 4). Since page P9 is not full, the last two records in the representation of Version 3 (namely, M3 and (V2, (15, 17), 11)) are stored into P9 after the last copied object, B1. The usefulness of page P9 is 41.7% because materializing its segment involves extra pages P6 and P4 and five objects out of these three pages are useful. So, P9 is useful and kept intact.

The complete version insertion algorithm is shown in Figure 5.

3.3. Complexity Analysis

Version Retrieval I/O Cost. Assume that the copy-based representation of version V is stored in pages P_1, P_2, P_3, $\cdots$, P_n and let S_1, S_2, S_3, $\cdots$, S_n, be the logical segments of these pages respectively. Since by construction all these pages are useful, materializing each segment, S_i, is bound by $1/U_{min} \times size(S_i)$. The I/O cost of materializing the complete version is the sum of the I/O cost of materializing segments S_1, S_2, S_3, $\cdots$, S_n, which is bound by:

$$1/U_{min} \times (size(S_1) + size(S_2) + \cdots + size(S_n))$$

where

$$size(S_1) + size(S_2) + \cdots + size_(S_n) = size(V)$$

Therefore, the version retrieval cost of version V is bound by: $size(V)/U_{min}$.

Minimizing Copying. Two further refinements are used in the version insertion algorithm to minimize unnecessary copying. To materialize P we have to access pages $P_1, ..., P_n$; but if P_1 is also accessed to materialize the page preceding P it can be excluded from the count of P. Furthermore, the materialization of the segment corresponding to P might write two or more pages. Then we might stop the materialization as soon as the first page boundary is reached, and return to step 2 at that point; the second page and the other pages might in fact be still useful.

Storage Cost. The copy-based $UBCC$ scheme consists of three kinds of records: actual object records, copied object records and reference records. Actual objects include new objects and updated objects. Since deleted objects are not removed from storage, they do not affect the size of the database. The new object part is bound by $O(S_{chg})$.

The number of objects that got copied once is bound by $U_{min} * S_{chg}$. Objects which got copied twice must be copied from those objects copied once already. Therefore, the total number of objects copied twice is bound by ${U_{min}}^2 * S_{chg}$. Similarly, the number of objects that got copied i times is bound by ${U_{min}}^i * S_{chg}$. Collectively, the total number of copied objects is bound by :

$$\sum_{i=1}^{\infty} {U_{min}}^i * S_{chg} \ = \ S_{chg} \times \frac{U_{min}}{1-U_{min}}$$

Finally, the number of reference records is the same as the number of changes. Therefore, the total size of reference records is :

$$S_{chg} \times K$$

where K denotes the average ratio between the size of the reference records and the document objects. As a result, the total size of reference records is bound by $O(S_{chg})$ as well.

Combining these three parts, the storage of the copy-based UBCC is $O(S_{chg})$, that is, linear in the size of changes in the version evolution.

4. Performance Analysis

We compared the performance of the copy-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 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. In the first set of experiments we used a document evolution with the following characteristics:

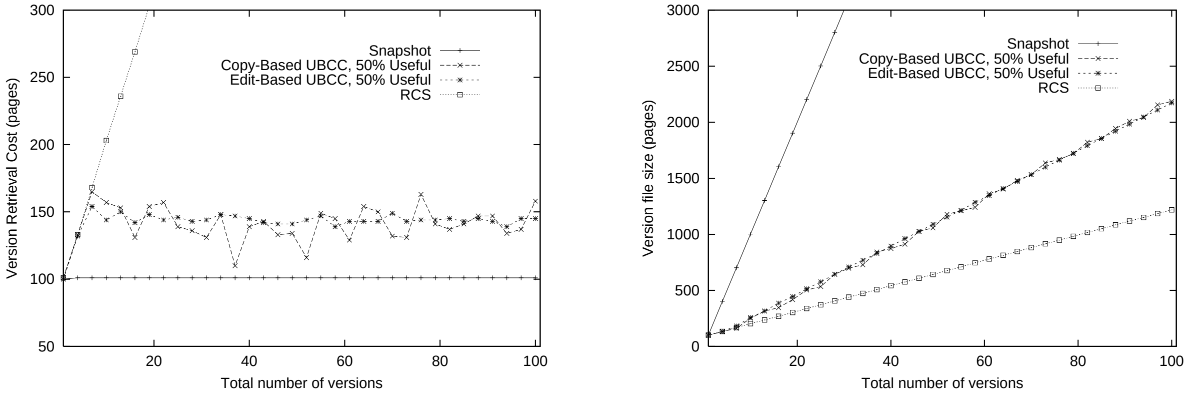

Figure 6. Version retrieval and storage cost with 50% usefulness requirement.

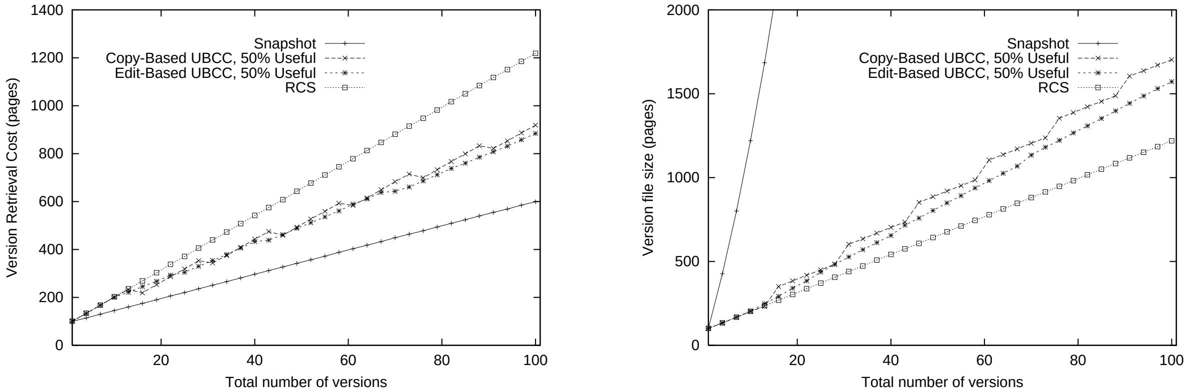

Figure 7. Version retrieval and storage cost with increasing document size.

- 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.

The other two sets of experiments evaluate how the schemes behave when the document grows or shrinks in size.

In the second set, insertions add up to 10% of the document size and deletions to 5% of document size (for a 5% of net growth). The third set has insertions for 5% of document size and deletions for 10% of document size.

Figure 6 shows the result of the first set. Version retrieval cost is measured as the number of page I/O's needed to reconstruct a version. The "Snapshot" scheme clearly 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 trade-off between these two extremes and deliver intermediate performance. The figure shows that the average retrieval cost for the usefulness-based schemes remain 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 copy-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 copy-based scheme shows more substantial fluctuations around the smoother line of the edit-based scheme. Thus, some valleys in the copy-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 copy-based and edit-based schemes are very close to each

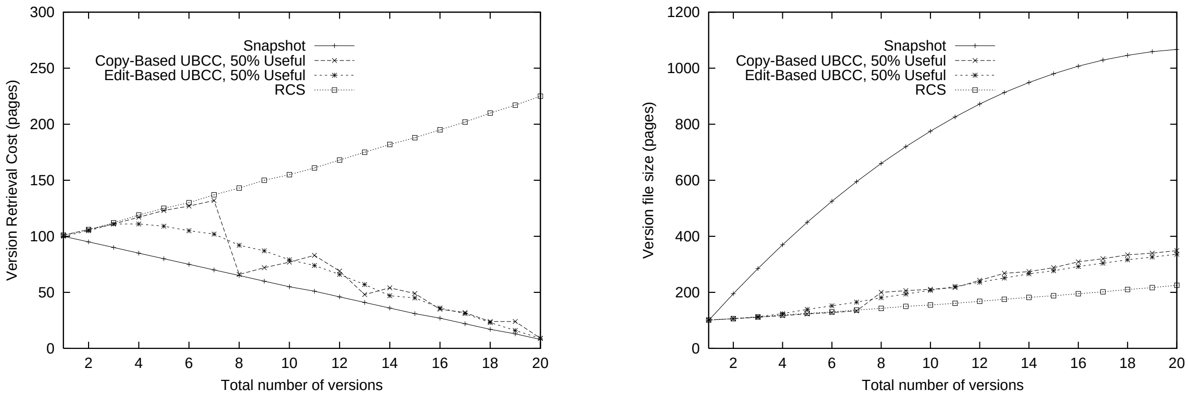

Figure 8. Version retrieval and storage cost with decreasing document size.

other, for the same usefulness factor.

Figures 7 and 8 show the results of the second and the third set. The copy-based scheme behaves similarly in both the increasing and the decreasing evolving size experiments. Its version retrieval cost closely fluctuates around the edit-based scheme, while their storage cost remain close to each other.

5. Conclusion

Due to the growing importance of versioned XML documents, we have been seeking strategies for optimizing their storage and retrieval. In a previous paper, we concentrated on edit-based representations and proposed a usefulness-based management technique (UBCC) that provides better overall performance and flexibility than more traditional version control methods such as RCS [2, 3]. As discussed in [3], the edit-based UBCC for multi-version documents achieves performance levels that are typically better than those obtainable using techniques developed for transaction-time databases and persistent objects managers.

In this paper we developed a copy-based representation technique that in terms of generality and flexibility of representation is superior to the edit-based representations favored by all previous authors. A main contribution of this paper has been to extend the usefulness based management to our new copy-based scheme, as to achieve the same level of performance on storage and retrieval as that obtained using the edit-based UBCC. Our copy based scheme stores and retrieves each version as a list of sublists without using edit scripts. This new scheme minimizes the version retrieval I/O overhead and offers the following advantages:

- changes such as document reorganization, and replication of selected document objects are supported along with the traditional insertion, deletion and updates supported by the edit scripts,
- list representation (unchanged segment records) are stored with actual objects, eliminating the need for a separate edit script. Only net effect of changes are used, and intermediate changes are factored out,
- multiple concurrent versions can be supported along with successive temporal versions.

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.

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

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

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

[7] 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.

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