The VLDB Journal (2002) 11: 332–353 / Digital Object Identifier (DOI) 10.1007/s00778-002-0079-4

Efficient schemes for managing multiversionXML documents

Shu-Yao Chien[1], **Vassilis Tsotras**[2], **Carlo Zaniolo**[1]

[1] Department of Computer Science, University of California, Los Angeles, California 90095, USA; e-mail: {csy,zaniolo}@cs.ucla.edu
[2] Computer Science Department, University of California, Riverside, California 92521, USA; e-mail: tsotras@cs.ucr.edu

Edited by Alon Y. Halevy. Received: December 15, 2001 / Accepted: June 1, 2002
Published online: December 19, 2002 – © Springer-Verlag 2002

Abstract. Multiversion support for XML documents is needed in many critical applications, such as software configuration control, cooperative authoring, web information warehouses, and "e-permanence" of web documents. In this paper, we introduce efficient and robust techniques for: (i) storing and retrieving; (ii) viewing and exchanging; and (iii) querying multiversion XML documents. We first discuss the limitations of traditional version control methods, such as RCS and SCCS, and then propose novel techniques that overcome their limitations. Initially, we focus on the problem of managing secondary storage efficiently, and introduce an *edit-based* versioning scheme that enhances RCS with an effective clustering policy based on the concept of page-usefulness. The new scheme drastically improves version retrieval at the expense of a small (linear) space overhead. However, the edit-based approach falls short of achieving objectives (ii) and (iii). Therefore, we introduce and investigate a second scheme, which is reference-based and preserves the structure of the original document. In the reference-based approach, a multiversion document can be represented as yet another XML document, which can be easily exchanged and viewed on the web; furthermore, simple queries are also expressed and supported well under this representation. To achieve objective (i), we extend the page-usefulness clustering technique to the reference-based scheme. After characterizing the asymptotic behavior of the new techniques proposed, the paper presents the results of an experimental study evaluating and comparing their performance.

Keywords: XML database – Version management – Historical queries – Temporal indexing – Temporal clustering

1 Introduction

The problem of managing multiple versions of XML documents is present in many applications [20] and poses new research challenges. Traditional application domains that rely on version management, such as software configuration and cooperative work, increasingly use XML for representing and exchanging information as they migrate to a web-based environment.

New application domains are also emerging for XML versioning; an important and pervasive one is the link permanence of 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, professionally managed sites and content providers will have to rely on document versioning. In fact, we might soon see 'e-permanence' standards established for critical web sites of public interest [36].

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 [14]. In fact, plans for global warehouses to preserve the complete history of the web are well under way [37].

Some of the problems occurring in multiversion documents are similar to those of transaction-time databases where object histories are maintained [15]. 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.

Several sophisticated schemes have been proposed for managing changes in semi-structured information [5], and structured documents [22,38]. Nevertheless, these approaches focus mainly on modelling version changes for individual objects without optimizing the efficiency of version retrieval. Version modelling is also the major concern in previous work on CAD and OODB version management schemes [2,12].

Two popular version management schemes have been developed for software configuration management [13], namely,

RCS [16] and SCCS [24]. RCS is *edit-based*: the most current document version is stored intact while previous versions are kept as *reverse* editing scripts. For any version except the current one, extra processing is needed to apply the reverse editing script and generate the old version. In a symmetric representation, RCS stores the very first document version and maintains future versions with a *forward* edit script.

Rather than appending version differences, SCCS [24] inserts editing operations in the original document and associates a pair of timestamps (or version ids) with each document segment (object) 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 schemes treat a document as a sequence of lines of text, and ignore the rich structure of documents, thus impairing their ability of supporting structural queries. Furthermore, they lack sophistication in their secondary storage management. Both RCS and SCCS may read document segments which are no longer valid for the retrieved (target) version, causing additional processing costs. For example, if we use RCS with the forward edit script, the total I/O cost to retrieve a version is proportional to the size of the first version plus the size of changes from this version till the retrieved one. For SCCS, the situation can be even worse: the whole version file needs to be read for any version retrieval. This cost is reduced by maintaining 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.

To efficiently support multiversion documents, better clustering techniques are needed. Our first scheme is *edit-based*; as such it enhances the original RCS with an effective page clustering approach (based on the notion of *page-usefulness*) that clusters the objects of any version in few data pages. The result is a drastic reduction (as compared with RCS) in version retrieval time, while the space used by our technique remains linear (like RCS) to the number of changes in the document evolution.

An additional requirement is for the versioned document to be easily exchanged at the transport level, across applications and to remote sites. However, the edit scripts used by our edit-based scheme represent a special object that cannot be easily accommodated at the transport level without XML extensions. The ideal solution is to represent the history of a versioned document as yet another XML document – this will turn web-browsers, style sheets, query processors, and the many great XML tools into a ready-made support environment for XML versions. To address this requirement we then propose a *reference-based* version management scheme that preserves the logical structure of the evolving document through the use of object references. By preserving the document structure, the scheme facilitates content-based querying. Furthermore, a modified clustering technique is used to preserve the scheme's efficiency at the storage level.

Two related works on versioning have recently appeared in [1,6]. In particular, [1] presents an archiving technique for scientific XML data. This approach is based on SCCS and clusters historical information by object (that is, all versions of the same object are physically clustered together). In contrast, our clustering scheme clusters objects by version. Moreover, [1] does not maintain the logical order of a given version and uses key-based id's for identifying the objects. Such ids are needed for clustering the new version of an object at the appropriate place. In contrast, our scheme does not depend on keys and maintains the logical document order. Chorel [6] presents a logical model for managing historical semistructured information. The Chorel model also clusters the historical information within each object.

The rest of the paper is organized as follows. Section 2 contains preliminary material and an example to be used throughout the presentation. Section 3 introduces the edit-based versioning scheme, while the reference-based approach appears in Sect. 4. In Sect. 5, we compare the performance of the edit-based scheme with the basic RCS and SCCS schemes, with multiversion B^+-trees [28], and partially persistent lists [31]. While these two techniques were proposed in the past in a different context, they are relevant to the problem at hand, insofar as they can be used to implement SCCS, while improving its performance in version retrieval and new version insertion. In Sect. 6, we compare the performance of the reference-based scheme with the edit-based scheme, since this proved to be the most efficient between the different schemes considered. Section 7 concludes the paper and discusses open problems for further research. Details on the multiversion B^+-tree and partially persistent lists appear in the Appendix.

The edit-based scheme was introduced in [7], and the reference-based approach was proposed in [44]. This paper adds several improvements and additions, including: (1) complexity analysis for the two schemes; (2) a detailed algorithm for full version materialization; (3) an improved representation for the reference-based approach; and (4) the results of new experiments on the performance of the SCCS scheme and the comparison of indirect versus direct references.

2 Representing XML documents

Various approaches have been recently proposed for representing and storing XML documents [32,39–42]. Typically, an XML document is represented as an ordered tree. To capture the tree structure a numbering scheme is needed. Alternative numbering schemes include:

1. Using Dewey's notation [9] which identifies nodes by their tree address starting from the root (e.g., element F in Version 1 of Fig. 1 has address 3.2); or
2. Using the tree preorder traversal numbers [9] (shown in Fig. 1) enhanced by the tree level [39–41]; or
3. Special representations such as the durable node numbers and ranges [40] (non-consecutive durable numbers can be assigned to nodes during the preorder traversal; then additional number is added to the node to bracket the node numbers of its children and children's children ... [43].)

For this paper we will use the second approach. Then a document element is represented by a triplet: (element-name, preorder#, tree-level).

As a motivating example consider the three consecutive versions of a document tree that appear in Fig. 1. Version 1 is then represented by the following list of triplets: $(A, 1, 0)$, $(B, 2, 1)$, $(C, 3, 1)$, $(D, 4, 1)$, $(E, 5, 2)$, $(F, 6, 2)$, $(G, 7, 2)$, $(H, 8, 1)$, $(I, 9, 1)$, $(J, 10, 1)$, $(K, 11, 1)$, $(L, 12, 2)$, $(M, 13, 2)$, $(N, 14, 1)$, $(O, 15, 1)$, and $(P, 16, 1)$. This list is

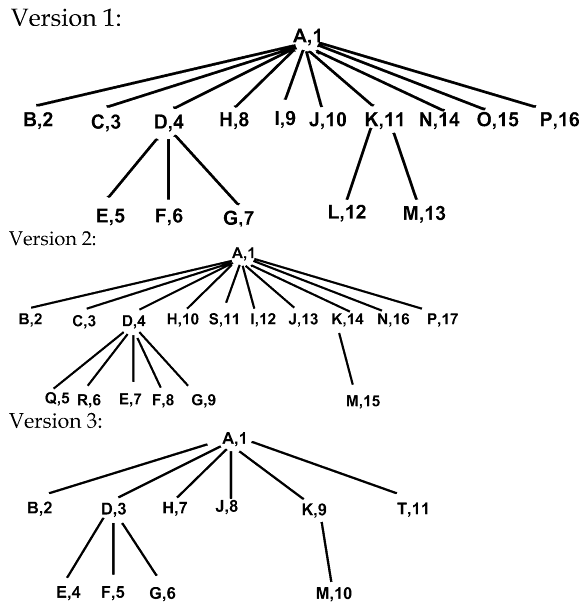


Fig. 1. Three versions of a document tree

ordered by the element preorder. The tree structure can easily be reconstructed from this list using the tree-level information. Similarly, Version 2 is represented by the list: $(A, 1, 0)$, $(B, 2, 1)$, $(C, 3, 1)$, $(D, 4, 1)$, $(Q, 5, 2)$, $(R, 6, 2)$, $(E, 7, 2)$, $(F, 8, 2)$, $(G, 9, 2)$, $(H, 10, 1)$, $(S, 11, 1)$, $(I, 12, 1)$, $(J, 13, 1)$, $(K, 14, 1)$, $(M, 15, 2)$, $(N, 16, 1)$ and $(P, 17, 1)$. As elements are added/deleted in the tree their preorder is changed accordingly. Finally, Version 3 becomes: $(A, 1, 0)$, $(B, 2, 1)$, $(D, 3, 1)$, $(E, 4, 2)$, $(F, 5, 2)$, $(G, 6, 2)$, $(H, 7, 1)$, $(J, 8, 1)$, $(K, 9, 1)$, $(M, 10, 2)$, $(T, 11, 1)$.

3 The edit-based approach

An advantage of using edit scripts is their simplicity: such scripts are typically created by an XML editor or by a package that computes the structured DIFF between two successive document versions [25]. Our first approach in creating an efficient scheme for multiversion documents is edit-based. We extend the traditional forward RCS scheme in two ways: (i) the document objects are separated from the edit script; and, (ii) a page clustering scheme is introduced. Because of (i), the script is rather small and easily accessed. Our clustering scheme (termed *usefulness-based copy control (UBCC)*) enables better clustering of document objects by version-id, which drastically improves I/O for version reconstruction. In the rest of the paper we will use the terms "edit-based scheme" and "UBCC" interchangeably.

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 100th version requires only 10% retrieval overhead. However, if 70% of the document changes between versions, then retrieving the 100th version could cost 70 times the effort of retrieving the first one! In this second case, storing complete version snapshots 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.

3.1 Usefulness-based clustering

For simplicity, assume that the document's evolution creates a linear sequence of 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. In our discussion, we use forward scripts which appeal to the intuition, since they represent the history of the evolution of the document.

Consider the actual document objects and their organization in disk pages. As new objects are added between versions, their records are stored sequentially in pages. Object deletions are not physical but logical. Records of deleted objects remain in the pages where they were recorded, but are marked as deleted (this marking is maintained separately by the edit script). An object update can be represented by a deletion followed by the insertion of the updated object. As the document evolution proceeds, various pages will contain many "deleted" objects and few, if any, valid objects for the current version. Such pages, will provide few objects for reconstructing the current version. As a result, a version retrieval algorithm may have to access many pages that contribute little to the target version. Ideally we would like to cluster the objects valid at a given version in a few, *useful* pages. Then the version retrieval becomes very effective by accessing only the useful pages for the target version. 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.

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 deleted, and at version V_4, object O_5 is deleted. 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's usefulness is 70% for version V_3 and drops to 55% 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, a page is considered *useful* for all versions for which its usefulness is above U_{min}. When a page's usefulness falls below U_{min}, the page is considered *useless* for that version and the records of all objects that are still valid in this page are copied [1] (i.e., salvaged) to another page (hence the

[1] This is similar to the time-splits used in temporal databases [11, 18, 26–28].

name UBCC). When objects are copied in new disk pages they preserve their relative document order.

The page usefulness definition needs to be extended for non-full pages. Since records are stored in pages sequentially, the last page written by a version may be non-full. By definition such non-full page is useful for the version that created it. This is needed since the page can still contain elements valid for this version.

The value of U_{min} is set between 0 and 1 and represents the main performance parameter of our scheme. For instance, if $U_{min} = 0.6$ (i.e., 60%), then page p becomes useless at Version 4; at this point objects O_1, and O_4 are copied to a new page. High U_{min} implies better clustering and thus faster version reconstruction (since the answer is clustered in fewer pages) but additional storage due to more copied records. Nevertheless, as will be shown in Sect. 3.3, the storage introduced by the above copying is proportional to $\frac{S_{chg}}{\sigma(1-U_{min})}$, where S_{chg} denotes the total size of document evolution (i.e., the information stored in RCS), and σ is the page size. Moreover, the number of useful pages for version V_i is bounded by $\frac{1}{U_{min}} \times \frac{size(V_i)}{\sigma}$, where $size(V_i)$ denotes the number of objects in document version V_i. Clearly, $\frac{size(V_i)}{\sigma}$ represents the minimum number of pages required to store version V_i when all its objects are clustered together. The $\frac{1}{U_{min}}$ factor is due to the fact that the objects of V_i are instead scattered in several pages. As will be shown from the performance evaluation, the page-usefulness clustering delivers the adaptable self-adjusting method that we were seeking: it combines the storage behavior of RCS with the retrieval performance of a complete version-snapshot scheme.

3.2 The edit script

The edit script is used for: (i) identifying the useful pages per version; and (ii) maintaining the document's logical order. We use the following notation when storing the script of version V_k: (i) the insertion of object X at (preorder) position i at level j, is represented in the edit-script by $ins(@X, i, j)$, where $@X$ is the location (i.e., page number and offset) where the record of object X is actually stored; and (ii) $del(i)$ denotes the deletion of the object that, without this deletion, would occupy (preorder) position i in V_k.

To describe the creation of the UBCC script consider the example in Fig. 1. Version 1 of the document is created by inserting sixteen new objects; these objects are stored in pages p1 through p4 while the corresponding script is stored separately as a sequence of ins operations (Fig. 2). For simplicity all objects are of the same size and a data page can hold four objects. Assume that U_{min} is set to 70%.

Version 2 is created by the changes: insert Q, R as children of D before E; insert S as a sibling between H and I; delete L and delete O. Its preorder appears in Fig. 1. The newly inserted document objects are stored (in relative order) in page p5 (Fig. 2). These insertions are represented in the edit script E2 as: ins(@Q,5,2), ins(@R,6,2) and ins(@S,11,1). Element Q is inserted as the new first child of element D; since D had preorder position 4 in Version 1, element Q will get position 5 in Version 2. Similarly, R gets position 6. Element S takes the next position after element H. In Version 1, H was in position 8, but with the two previous additions, it moves to position 10 in Version 2; hence S gets position 11. After the insertion of S, element L would have moved to position 15 in Version 2. Therefore its deletion is represented as del(15) in script E2. After deleting L, element O moves to position 17 and its deletion results to del(17). Despite the two deletions, pages p3 and p4 are still useful (75%) for Version 2, as are pages p1, p2 and p5 (which is not full but is the last page created by Version 2).

Version 3 is generated by the changes: delete C, Q, R, S, I, insert T as a sibling between K and N, and delete N, P. The deletions make pages p3, p4 and p5 useless for Version 3, since their usefulness falls below U_{min} (to 50%, 25%, and 0%, respectively). Hence, their valid objects for Version 3 – namely, J, K and M must be copied. The newly inserted object T and the copied objects J, K, M are stored into a new data page p6 in their relative document order in Version 3. For each copied object, a pair of entries – one deletion followed by one insertion – are added to the edit script. The first five del operations in script E3 correspond to the deletions of C, Q, R, S, I (Fig. 2). Then del(8) and ins(@J,8,1) represent the copying of object J from page p3 to p6; note that @J in script E3 refers to page p6. Similarly, del(9), ins(@K,9,1) and del(10), ins(@M,10,2), correspond to copying objects K, M, respectively. The copying procedure effectively clustered the objects valid for Version 3 into useful pages p1, p2 and p6.

Version reconstruction. A given version is reconstructed by first visiting the edit scripts to identify the objects valid for this version (in their appropriate document order). Then, the data pages containing the actual objects are retrieved. Edit scripts are processed in a *gap-filling* fashion. Consider reconstructing

```
                  VERSION 1
---------------------------------------------
     Data Pages          UBCC Script E1
---------------------------------------------
   +------------+    ins(@A,1,0), ins(@B,2,1),
p1 | A, B, C, D |    ins(@C,3,1), ins(@D,4,1),
   +------------+    ins(@E,5,2), ins(@F,6,2),
p2 | E, F, G, H |    ins(@G,7,2), ins(@H,8,1),
   +------------+    ins(@I,9,1), ins(@J,10,1),
p3 | I, J, K, L |    ins(@K,11,1), ins(@L,12,2),
   +------------+    ins(@M,13,2), ins(@N,14,1),
p4 | M, N, O, P |    ins(@O,15,1), ins(@P,16,1).
   +------------+

                  VERSION 2
---------------------------------------------
     Data Pages          UBCC Script E2
---------------------------------------------
   +------------+    ins(@Q,5,2), ins(@R,6,2),
p5 | Q, R, S    |    ins(@S,11,1), del(15), del(17).
   +------------+

                  VERSION 3
---------------------------------------------
     Data Pages          UBCC Script E3
---------------------------------------------
   +------------+    del(3), del(4), del(4),
p6 | J, K, M, T |    del(8), del(8),
   +------------+    del(8), ins(@J,8,1),
                     del(9), ins(@K,9,1),
                     del(10), ins(@M,10,2),
                     ins(@T,11,1),
                     del(12),del(12).
```

Fig. 2. The edit-based scheme

Version 3. The first entry in script E3 is del(3) denoting that the object in the third position must be deleted; moreover this entry also denotes a *gap*, to be filled with objects from previous versions. To fill this gap, the first three objects of Version 2 are thus requested. The first entry in E2 is ins(@Q,5,2) which corresponds to inserting an element in the 5th position, i.e., this is another gap. Thus the objects needed are retrieved from the script of the previous version, E1. Entries ins(@A,1,0), ins(@B,2,1) and ins(@C,3,1) of E1 are returned to Version 2 and recursively to Version 3. Entry ins(@C,3,1) is then nullified by the del(3) operation in E3. Since @A and @B correspond to page p1, this page is retrieved and the actual objects A,B are found. The third object of Version 3 is retrieved by issuing another next-object operation to Version 2 and recursively to Version 1. As a result ins(@D,4,1) from E1 is returned to Version 3. A subsequent next-object request to Version 2 will return entry ins(@Q,5,2) (the gap in E2 has been filled) which is then nullified by the del(4) entry in E3. Similarly, ins(@R,6,2) in E2 is nullified by the next del(4) in E3. The fourth object of Version 3 will be ins(@E,5,2) from E1 (since ins(@S,11,1) from E2 creates a new gap). This gap-filling procedure continues through the script E3 until all objects of Version 3 are retrieved. The version reconstruction algorithm is shown in Fig. 3.

The reconstruction algorithm only retrieves pages that are useful for the version being materialized. For example, object J in Fig. 2 is retrieved from page p3 for Version 2, but from page p6 for Version 3. This is because page p3 is not useful for Version 3, whose live objects were copied to page p6 – via the deletion and reinsertion of object J in the E3 script. In the example of Fig. 2, objects of Version 3 are stored in pages p1, p2, and p6 sorted in their natural order. Thus, to reconstruct Version 3, we only need to scan these pages sequentially, requiring only one page of main memory as a buffer. However, this property does not hold in general. For instance, say that our Version 3 also contains objects Q and R from Version 2. (The UBCC script in E3 must then be modified by removing two `del(8)` entries and increasing the second argument of the remaining entries by two.) Then, the materialization of Version 3, involves taking: (i) B, D, from p1; (ii) E, F, G, H from p2; (iii) J, K, M, from p6; (iv) Q and R from p5; and (v) returning to p6 to fetch T. Hence, to retrieve the elements of any version in total order, a sort-merge process is needed among its useful pages. In general, the objects of a version V_n may be stored in (useful) pages generated in k previous versions. (Frequently, useful pages for version n are contained in versions $V_{n-k}, \ldots, V_{n-1}$; but the situation where some intermediate versions are not contributing is also possible – e.g., Version 2 is not contributing to Version 3 in Fig. 2.) For typical situations, the value of k is small (say below 20); then, a buffer of size k suffices to reconstruct V_n in one pass through its useful pages. An external sorting algorithm could instead be used if there were no sufficient memory to hold k pages.

Edit script snapshots. The reconstruction of version V_i may involve reading all edit scripts from E_i to E_1. While edit scripts are much smaller than the actual document data, as the number of versions grows, their overall size will accumulate and may affect the version retrieval efficiency. To solve this potential problem, whenever the size of the edit scripts needed for

```
RECOVER(UBCC_SCRIPT S, INT request) {
 count = 0;
 while (count < request) {
     if (s is out of edit operation)
     {
       gap = request - counter;
       RECOVER(UBCC script of previous
               version of S, gap);
       Append returned object list.
       count = count + gap;
     }
     else if (target position < the position
              of current operation)
     {
       gap = position of current operation
             - target position;
       RECOVER(UBCC script of previous
               version of S, gap);
       Append returned object list.
       count = count + gap;
     }
     else if (current operation is delete)
     {
       RECOVER(UBCC script of previous
               version of S, 1);
       Nullify the retrieved object.
     }
     else if (target position = the position
              of current operation) {
       Append the object of current operation
         to return_object_list.
       count++;
     }
 }
 return return_object_list;
}
```

Fig. 3. Version retrieval algorithm for the edit-based scheme

reconstructing a particular version gets over a threshold (e.g., 10% of the size of the current version) an *edit script snapshot* is built for this version. This snapshot contains one insert record for each object in the current version. The edit script $E1$ in Fig. 2 is an example of an edit script snapshot. The reconstruction of a version only requires reading the last snapshots and the edits after that. As we shall see in the next section, the length of these two is a small percentage of the length of the current version. In addition, observe that the snapshot policy we are using tends to generate infrequent snapshots when the document grows (and snapshots are least effective), and frequent ones when the document is shrinking and snapshots are most effective.

3.3 Complexity analysis

The overall storage of the edit-based approach consists of two parts: (i) the space used by the data pages (as managed by the UBCC); and (ii) the space for storing the edit scripts.

Cost of the UBCC scheme. Data pages store new objects and objects copied by the UBCC clustering. New objects correspond to either newly inserted objects or updated objects. Since deleted objects are not removed from storage, deletions do not affect the size of data. Let S_{chg} denote the total size of the document evolution (without any usefulness-based copying – i.e., as in the RCS scheme). The total size of objects that are copied once is $U_{min} \times S_{chg}$. Objects that are copied

twice must be copied from those objects that have already been copied once. Therefore, the total size for objects copied twice is $(U_{min})^2 \times S_{chg}$. Similarly, the storage used by objects that were copied i times is $(U_{min})^i \times S_{chg}$. Collectively, the total storage used by copied objects is:

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

Therefore, in the UBCC scheme the total storage used for the original objects plus their copies is $\frac{S_{chg}}{1-U_{min}}$, whereas the storage used by the RCS scheme is S_{chg}.

Edit script cost. Clearly, too frequent edit script snapshots might cause an excessive use of storage while too infrequent ones might incur large retrieval overhead. Our improved 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, $0 < k < 1$ is selected to keep the size of the script a small percentage of the overall storage. In our experiments, we have kept the value of k around 0.1.

Lemma. The total size of the edit script with snapshot is proportional to S_{chg}.

Proof. Let E denote the size of the whole edit script (without snapshots). Since each version change is represented by an entry in the edit script, E is proportional with respect to S_{chg}. Thus, it suffices to show, that the size of the edit script snapshots is linear in E, the size of the whole edit script (without snapshots).

Assume that an edit script snapshot is generated for version V_m while the next edit script snapshot is for version V_n, where $m < n$. Let $\delta E_{(m,n)}$ denote the size of the edit scripts from V_m to V_n, and let T_n denote the snapshot of V_n taken when $\delta E_{(m,n)}$ has just surpassed $k \times size(V_n)$. That is, a new snapshot is taken whenever

$$\delta E_{(m,n)} \geq k \times size(V_n)$$

i.e., as soon as:

$$size(V_n) \leq \frac{1}{k} \times \delta E_{(m,n)}$$

Now, let k' denotes 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, we obtain:

$$size(T_n) \leq \frac{k'}{k} \times \delta E_{(m,n)}$$

That is, the size of the snapshot T_n is linear in the size of the edit scripts between versions V_m and V_n. Summing up the above inequality for all snapshots generated, we obtain that the size of all snapshots is proportional to E by a constant (k'/k). Typical values for k' and k are, respectively, 0.05 and 0.1; therefore, the snapshots here add a 50% overhead to edit scripts, which still remains a small percentage of the size of the UBCC data pages. Furthermore, the cost of searching the edit script is guaranteed to be a small percentage of that of reconstructing the version – a property that would be lost without snapshots. □

4 The reference-based scheme

The edit-based approach discussed in the previous section provides an efficient scheme for storing multiversion documents and reconstructing any version of such documents. An advantage of this scheme is its incremental nature whereby each new version is added to the existing repository, without changing the information previously stored. However, the edit-based approach suffers from several limitations. For instance, revisions where the document is reorganized and various elements are moved to new positions cannot be easily represented under the edit-based scheme. In addition, the fact that the information is split between the actual database and the script complicates the task of supporting queries. Finally, the scheme is not suitable for external representations where multiversion documents can be viewed by users through browsers, or need to be exchanged between sites. In fact, edit scripts create a special object that cannot be easily viewed as an XML object. Ideally, we would like the history of a versioned document to be represented as yet another XML document: this will turn web-browsers, style sheets, query processors and the many great XML tools into a ready-made support environment for XML versions.

The *reference-based versioning scheme (RBVM)*, discussed next, overcomes the limitations of the edit-based approach, while retaining its benefits in terms of efficient storage management.

While edit-based approaches focus on representing changes, the new scheme concentrates on representing the document 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 objects of the following two kinds:

- *Reference records* which denote maximum common segments shared between the new version and the previous version, and
- *Actual document object records*.

The $RBVM$ scheme preserves the structure of the original document. Nevertheless, elements of Version V_j that have remained unchanged with respect to Version V_{j-1} are represented in V_j as references to the corresponding elements in V_{j-1}. The scheme can be illustrated by using the graphical representation used in the XPath specifications [19], in which XML documents are modelled as ordered-trees containing various types of nodes as shown in Fig. 4.

To simplify the discussion we only consider: *element*, *text*, *attribute*, and *reference nodes*. Then, a multiversion document can be represented as an ordered forest of XML-document

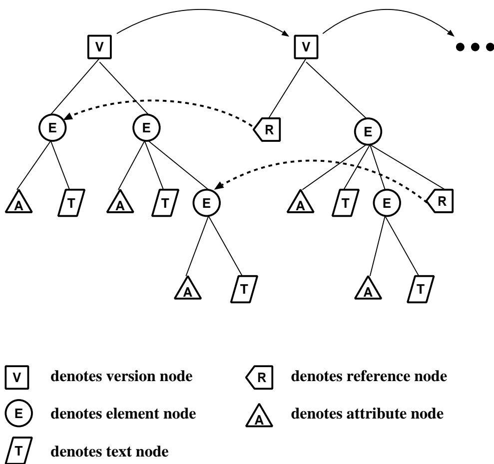


Fig. 4. The $RBVM$ tree model

trees, with references pointing from one version to the preceding one as shown in Fig. 4.

Example. Three successive versions of a book are shown in Figs. 5 and 6.

The initial version, Version 1, contains two chapters – "Introduction to TSQL2", and "TSQL2 Tutorial", and each chapter contains one and two sections, respectively. The initial version is always fully materialized. Version 2 is generated by modifying Version 1 with the following changes :

- DELETE the "TSQL2 Tutorial" chapter;
- INSERT "A Second Example" chapter after the first chapter.

Since the first chapter of Version 2 is the same as the first chapter of Version 1, a reference record is used in Version 2 to represent that unchanged element instead of storing the actual content. The reference record refers to the first element at the first level of Version 1, which has remain unchanged between the two versions. We use the node preorder numbers as their IDREF; thus this node can be referred to as $V1.1$. (However, '$V1$' need not to be stored explicitly since we are now materializing version $V2$, whose references only point to the previous versions '$V1$'.) The second chapter of Version 2 is a new chapter and, thus, is fully materialized and stored locally. Last, a link is built from Version 1 to Version 2. Note that deletions do not require any explicit representation; they are simply not listed in the new version.

Version 3 is generated from Version 2 via the following changes :

- UPDATE the textual content of the "A Second Example" chapter;
- COPY the "Concepts" section and insert it after the "Test Data" section under the "A Second Example" chapter.

As shown in Fig. 6, the "Introduction to TSQL2" chapter remains unchanged, thus, is represented by the reference ($V2.1$). Observe that *reference records of a version always refer to its previous version, which in turn might refer to its previous version*. Therefore, reference records are $logical$ and may be $indirect$.

The second chapter is $changed$ because its textual content is updated and a new "Concept" section is copied and inserted under it. Therefore, this chapter contains four reference nodes, each referring to corresponding elements in the previous version. Note that the last reference record is also indirect because it refers to a sub-element of an element represented by a reference record. Finally, the forest of the successive versions can be a linked together in a list (shown as a link from Version 2 to Version 3).

The references used by RBVM to shared parts with previous versions is reminiscent of the version sharing supported in EXODUS [4] and the Overlapping B-trees [3]. In these approaches a B+-tree is used to represent the physical storage of a large object and subsequent versions share common paths of this B+-tree with the previous version. In contrast, the RBVM scheme applies to any tree structured complex object. Furthermore it does not assume any physical representation of the versioned object while at the physical level it supports the notion of usefulness for better version clustering.

Restructuring and duplicating. It is often the case that two sections are switched in a new version. In addition, some passages and footnotes might be repeated at various points in the document. Our reference-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.

4.1 Version storage and materialization

We will now describe the actual storage of the $RBVM$ scheme by the same example used for the edit-based scheme. As shown in Fig. 7, we store two kinds of records: (i) content records, such as $(B, 2, 1)$ in Version 1 and $(Q, 5, 2)$ in Version 2; and (ii) reference records, such as $(5 : 8, 7, 0)$ in Version 2 or $(7 : 10, 5, 0)$ in Version 3. An entry of the form $(i : j, n, h)$ denotes that elements i through j in the previous version are copied to the current version, where the old element i now is moved to position n, for a shift of $n-i$ in the preorder traversal number. A level change is also possible and is denoted by h: thus, the element n in the new version has level $s + h$ where s is the level of element i in the old version. Therefore, record $(5 : 8, 7, 0)$, in Version 2, references elements 5 through 8 in Version 1. When Version 2 is materialized, the four elements of Version 1 with respective preorder numbers 5, 6, 7, and 8 are now given positions 7, 8, 9, and 10, respectively. Since $h = 0$ there is no shift in level from the previous copy.

In fact, all entries in Fig. 7 show a vertical shift of 0; this is because we are using the same example as in the edit script which cannot represent the relocation of elements to different levels. Such restructuring is, however, available in $RBVM$: For instance, if we replace $(5 : 8, 7, 0)$ by $(5 : 8, 7, 1)$, then `E`, `F`, and `G` are moved down to become children of `R`, while `H` is moved down to become the last sibling of `R`. (Such document restructuring can be easily produced by structured editors and XML documents update languages [32].) Since only references from a version to the previous version are allowed in $RVBM$ scheme, we do not need to include the version number in our reference records.

Note that, in Fig. 7, we have simplified our example by making some unrealistic assumptions about the size of the objects involved. In real-life situations, many more records than

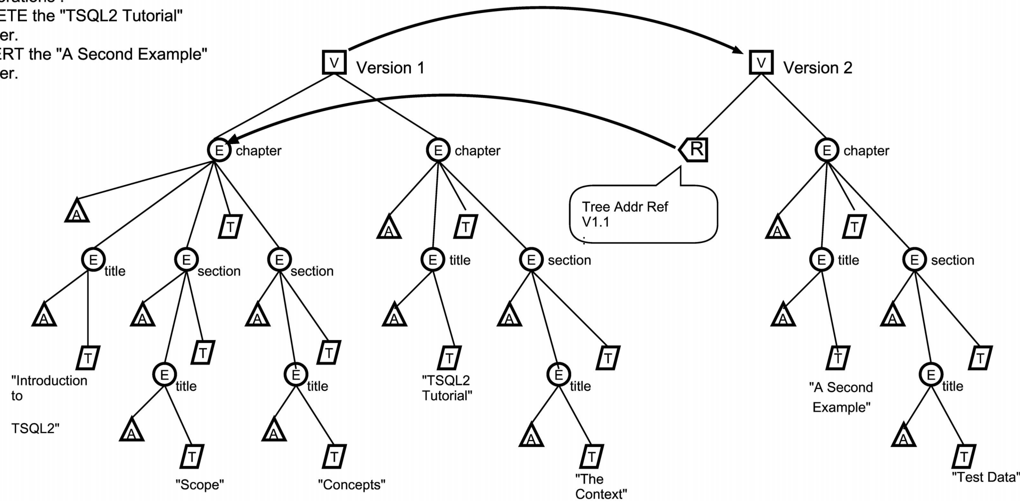


Fig. 5. Version 1 and version 2 of a book in $RBVM$

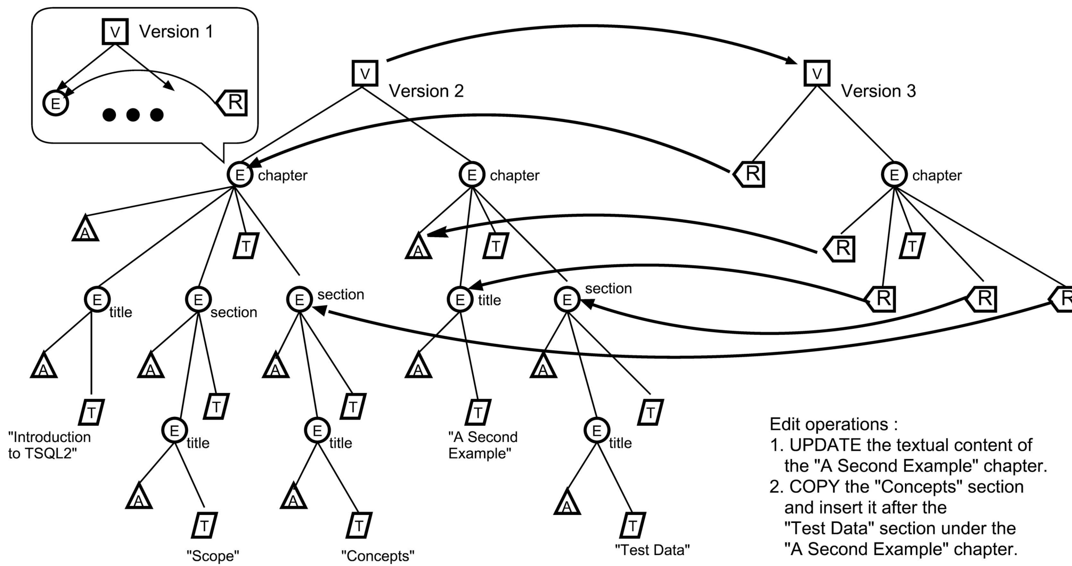


Fig. 6. Version 2 and version 3 of a book in $RBVM$

```
---------------------------VERSION 1------------------
p1:  (A,1,0), (B,2,1), (C,3,1), (D,4,1),
p2:  (E,5,2), (F,6,2),(G,7,2), (H,8,1),
p3:  (I,9,1), (J,10,1), (K,11,1), (L,12,2),
p4:  (M,13,2), (N,14,1), (O,15,1), (P,16,1).
-------------------------- VERSION 2------------------
p5:  (1:4,1,0), (Q,5,2), (R,6,2), (5:8,7,0),
p6:  (S,11,1),(9:11,12,0), (13:14,15,0), (16:16,17,0).
--------------------------VERSION 3------------------
p7:  (1:2,1,0), (4:4,3,0), (7:10,5,0),(13:15,8,0),
p8:  (T,11,1).
```

Fig. 7. The $RBVM$ scheme for the versions in Fig. 1

four fit in one page, different records are of different size, (e.g., reference records are typically much shorter than the content records) and a variable number of records fit in one page. The simplified example, however, captures the representation and reconstruction issues involved.

Version materialization. Consider now $(7 : 10, 5, 0)$ in Version 3: this denotes that starting at position 5 in Version 3 we copy elements 7 through 10 from Version 2 with zero shift in level. As we follow the references and we look for elements 7 through 10 in Version 2, we find that we must follow the references and fetch the actual elements from Version 1. In general, to materialize Version J, we need to use a recursive procedure that starts at Version J and then moves to the previous versions following the references.

In Fig. 8, we describe the function `produce` used to materialize any given Version `K` from element `Start` till element `End` (where the elements are denoted by their preorder

```
produce(K, Start, End, dx, dy)
      { if Version K contains an entry (I:J, N, L) with I<Start<=J
            then
              {Result:= produce(K-1,Start,min(J,End), N-I, L);
               Start:= J+1;}     /*  straddling */
             else Result:= empty;
        for each entry E in Version K where Start<= pos(E) <=End
           {if E=(I:J, N, L)
               then Result:= Result||produce(K-1, I, min(J,End), N-I, L)
                        /* since this is a reference element */
                else Result:= Result||E;
                        /* since E is an actual element */
           }
         return shiftxy(Result, dx, dy)
        }
```

Fig. 8. Version materialization for the RBVM scheme

numbers). Thus, to materialize a complete version of N elements, we simply issue a call produce(K, 1, N, 0, 0). The two zero arguments denote that there is no shift in the position and level of the retrieved elements since we want to retrieve the tree 'as-is'. (However, in general, by setting these arguments to the proper values it is possible to place the results returned by produce in a particular position of the tree being constructed.) The call produce(K, 1, N, 0, 0) expands all the reference records into actual elements, returning a representation just like that of Version 1. From this representation, a standard XML document is easily produced by eliminating the preorder numbers and replacing level numbers by nested XML tags. The function produce uses the following auxiliary procedures: (i) A||B denotes the result of appending B to A; and (ii) shiftxy(Result, dx, dy) changes each entry (a, b, c) in Result to (a, b + dx, c + dy).

Example. To materialize Version 3, we call produce(3, 1, N, 0, 0). The first entry is (1 : 2, 1, 0). Neither this (nor any other entry) satisfies the first condition. Hence we go directly to the second one where we call produce(2, 1, 2, 0, 0) (since there is no shift of position or level for the first elements from Version 2 to Version 3). Now, produce(2, 1, 2, 0, 0) finds (1 : 4, 1, 0) in Version 2, thus a call produce(1, 1, 2, 0, 0) is generated on Version 1. This call on Version 1 returns the first two elements A and B to Version 2, which then returns them to the original call on Version 3. Now Version 3 is ready to materialize its next entry: (4 : 4, 3, 0) by produce(2, 4, 4, −1, 0). This call starts from the first entry in Version 2; we are in the straddling situation whereby the call produce(1, 4, 4, 0, 0) is issued on Version 1. The results are then returned to Version 2, which then returns them to Version 1. The materialization of this version then resumes by materializing its third entry, and so on.

Therefore, versions are reconstructed in depth-first (from the last version to the previous ones) in a left-to-right fashion. Assume that Version K uses elements from previous versions up to $K - H$. Then, H will be called the depth of the history of Version K. In materializing Version K using the recursive procedure of Fig. 8, at most one page from each of the previous H versions is needed at any time. If the main memory buffer holds at least H pages, then each page containing data for Version K is accessed only once during the materialization of Version K. Hence this algorithm assures optimal secondary store retrieval performance for a buffer of size H. Usefulness-based techniques discussed in Sect. 4.4 can be used to limit the size of H. Furthermore a similar approach can be used to answer a variety of queries, which are discussed in the next section.

Condensed and regular representations. In the example of Fig. 7, we have assumed that reference records describe maximal unchanged segments between versions. We will refer to this representation as a *condensed* $RBVM$ scheme. A more conservative representation requires the reference records to refer to single elements of the previous version (including all their subelements). This is called a *regular* $RBVM$ scheme. For instance, the examples in Fig. 5 and 6 use a regular $RBVM$ representation. While condensed representations might have a better performance at the storage level, regular $RBVM$ representations preserve the logical structure of the document – as discussed in Sect. 4.3. The query and version materialization algorithms apply identically to condensed and regular representations.

4.2 Queries on versioned documents

There has been much interest in query languages for XML and semistructured documents [33]. Our focus here is to evaluate the effectiveness of our $RBVM$ model to support the basic queries that arise naturally for versioned XML documents. These include queries on document history and evolution – in addition to the usual content-based queries on version instances, and a combination of the two, as summarized by the following taxonomy:

- *Temporal selection.* This basically retrieves either a particular version of the whole document in its entirety (snapshot query), or successive versions of the same – e.g., retrieve Versions 9 to 17. The algorithm in Fig. 8 can efficiently support this query.
- *Document evolution & historical queries.* These queries focus on the changes between successive versions of the document. They include queries such as, "What's new in

Version 5 of this document?" The computation of structured diff discussed in [5] is another example.

- *Structural projection.* Assume that the user requests certain elements, such as the table of contents and the second chapter from our book. Normally, users want to see all the subelements of a given item – e.g., for Chapter 1, the user would like to see the sections, subsections, and paragraphs. We can use a notation similar to the one we have used for the $RBVM$ to represent structural projection queries. For example, if the second chapter of the book starts at position 533, level 5, and contains 100 elements we can represent it as follows: $(533 : 632, 1, -5)$. This returns a document that only includes the second chapter, as denoted by a position 1 and a resulting level of 0 which is the root level. When several elements are to be retrieved we can use the position and level parameters to arrange them into lists, forests or trees. Structural projection is a key ingredient in many queries involving temporal selection ("Show Chapter 1 for Versions 15 to 32."), history queries ("Show the history of changes for this document's abstract"), and content-based queries.
- *Content-based selection* This retrieves all versions that qualify the predicates in the WHERE clause of the query. Content-based selection often occurs in queries that also include structural selection, temporal selection, and even document history.

Structural projection. We assume that a Projection List, $\mathtt{PL(K)}$ is given, consisting of a sequence of non-overlapping reference entries from Version K, where entries have the form of reference records $(\mathtt{I} : \mathtt{J}, \mathtt{N}, \mathtt{L})$. Thus, in addition to extracting some entries, these entries can also be shifted to a new position and level (e.g., as needed to generate a new XML document). In relational query processing, it is often useful to allow the introduction of constant columns [34]. We can do the same here and allow $\mathtt{PL(K)}$ to contain content triplets, $(\mathtt{Element}, \mathtt{Position}, \mathtt{Level})$, along with the reference triplets. Then, it is clear that our $RBVM$ scheme can simply be viewed as a sequence of successive structural projection operations. Because of this similarity, we can then use the recursive procedure of Fig. 8 for implementing structured projections.

The simplest case is when we only need to compute a structural projection on a given Version K. Then, the algorithm basically reduces to:

```
sproject(K)
  {for each (I:J, N, L) in PL(K)
            out(K):= out(K)||
            produce(K, I, J, N-I, L)}
```

(For simplicity, we have assumed that $\mathtt{PL(K)}$ only contains reference entries: the generalization to the case where there are also constant entries is trivial.)

The more general case is when we want to materialize certain segments of the document for all the versions between K and M, with $K < M$.

The parts of the document we want to materialize could again be the TOC and Chapter 2. However, elements are inserted and deleted between versions, and may change the preorder number of the elements. Thus, a separate projection list $\mathtt{PL(K)}, \ldots, \mathtt{PL(M)}$ is needed for each version.

Even so, much of the work done in materializing $\mathtt{PL(I)}$ can normally be used to materialize $\mathtt{PL(I+1)}$. Thus, rather than repeating the materialization for each version, we would like to amortize the cost by, say, materializing $\mathtt{PL(K)}$ first and then, store the result of this operation and use it in materializing $\mathtt{PL(K+1)}$; then, after storing the result of $\mathtt{PL(K+1)}$ we proceed by materializing $\mathtt{PL(K+2)}$, and so on. This bottom-up materialization approach is far from optimal for two main reasons. The first is that the whole result for a version must be materialized, before it can be used for the next version. The second, and more serious problem, illustrated by the example below, is that $\mathtt{PL(I)}$ often does not contain all elements requested in $\mathtt{PL(I+1)}$. Then, as a result, the pages that have been read once for $\mathtt{PL(I)}$ may need to be read from the secondary storage again for those elements which do not appear in $\mathtt{PL(I)}$ but are requested in $\mathtt{PL(I+1)}$.

Example. Assume that we have a structural projection query requesting the second section and the third section of Chapter A, and elements of Chapter E. In addition, say that version 11 is obtained from version 10 by deleting the first section of Chapter A. Then $PL(10)$ contains Sects. 1 and 2 from Chapter A, but not Sect. 3, which then becomes Sect. 2 for $PL(11)$. Thus, if these three sections fall in the same page, we end up reading it once for $PL(10)$ and again for $PL(11)$, since the pages of chapter E replaced the old pages in the buffer. To avoid this problem, we have to infer that Sect. 3 of chapter A is needed by later versions and fetch and preserve it when constructing $PL(10)$.

To avoid these problems and achieve optimality with respect to secondary store retrieval, we propose a top-down algorithm to evaluate projection list requests. For a projection list PL the evaluation starts from the newest version, say Version $(K+1)$. When evaluating $\mathtt{PL(K+1)}$, the $RBVM$-represented Version $(K+1)$ is checked to locate target elements. If target elements are stored in Version $(K+1)$ they are simply returned. However, if the target elements are represented as a reference record, (S, E, dx, dy), a request $\mathtt{materialize(K, S, E, dx, dy)}$ is invoked to ask for the target elements from the previous version, Version K. The request $\mathtt{materialize(K, S, E, dx, dy)}$ is processed by first subdividing $\mathtt{PL(K)}$ in three sublists $L1$, $L2$ and $L3$, where $L1$ contains the entries of $\mathtt{PL(K)}$ that precede S, $L2$ the entries that are contained in $S : E$, and $L3$ the entries that follow E. This operation might require splitting entries in $\mathtt{PL(K)}$; e.g., an entry $(I : J, x, y)$ where $I < S$ and $E < J$ will be split in the three entries $(I : S, x, y)$, $(S : E, x, y)$, and (E, J, x, y). The purpose of splitting $\mathtt{PL(K)}$ into three segments, $L1$, $L2$ and $L3$, based on S and E is to make $L1$ and $L2$ materialized as a side effect of evaluating $\mathtt{materialize(K, S, E, dx, dy)}$. That is, those Version K data pages read for $\mathtt{materialize(K, S, E, dx, dy)}$ are also used to materialize $L1$ and $L2$ at the same time. Therefore, when $\mathtt{materialize(K, S, E, dx, dy)}$ is done those data pages read from Version K will never be needed for the rest of the evaluation, and we avoid multiple read of those pages. The same side effect is recursively applied to all earlier versions. That is, when $\mathtt{materialize(K, S, E, dx, dy)}$ is done, the pro-

jection list for *all* earlier versions have been processed to a certain point where no previously read data page is needed for any later version. Thus, the problem of multiple read of the same page is solved. Now we can define `materialize` as shown in Fig. 9.

In Fig. 9 we have use two auxiliary procedures: `split` to subdivide `PL(K)` in three disjoint sublists, and `extract` to extract from `Temp` the segments defined by $L2$. $L3$ contains the elements of `PL(K)` which will be processed by later calls.

Finally, the procedure `produce1` of Fig. 9 can be constructed from the procedure `produce` in Fig. 8 by taking the calls "`produce(K-1,I,min(J,End),N-I,L)`" in its body and replace them with "`materialize(K-1,I,min(J,End),N-I,L)`". Thus `materialize` calls on `produce1` (on the same Version K), and `produce1` calls on `materialize` on the previous Version $K-1$.

Therefore, the general computation for structured projection, for copies between Version M and Version N ($M \leq N$) is from top down as follows:

```
{for (K:=N; K>= M; K:=K-1)
     sproject(K)}
```

Because of our depth-first left-to-right processing, most of the actual work will be performed by the call on the last version: `sproject(N)`. In fact, this call will complete the processing when the projection lists of the various versions are identical. However, the calls to the previous versions are needed to finish up the processing when the projection lists are different.

Evolution history retrieval. Generating evolution history upon users' request is an important feature of version management systems. A typical query could be: *Retrieve the differences between Version M and its previous Version.* Typical algorithms [23] for computing differences between two structured documents all share a two-phase strategy: the matching elements in the two versions are first found, and then, the edit script is constructed from that. The first phase is computationally expensive, while the second phase only requires a bottom-up, breadth-first search on the two versions. With the $RBVM$ scheme, the first phase is no longer necessary, since that information is already embedded in the references (each reference denotes a matching segment).

The $RBVM$ representation of version J fully describes the difference between this version and the previous one. Thus, from a scan of version J we can easily compute:

1. The new elements inserted. For example, the last entry in Version 3 of in Fig. 7 is (T,11,1) denoting that element T was inserted in preorder position 11 at tree level 1;
2. The elements copied. For instance, (13:15, 8, 0) in Version 3 of Fig. 7 denotes that elements, 13, 14, and 15 have now been copied. The same elements can be copied more than once, and copied elements can be rearranged in a different order. For instance, if we want to reverse the order of those three copied elements we only need to write (15:15, 8, 0), (14:14, 9, 0), (13:13, 10, 0);
3. The elements deleted simply corresponds to the holes in the list of the elements copied. For instance, in Version 3 of Fig. 7, we find that elements 1, 2, and 4 of version 2 are copied, but element 3 has not been copied. Thus element 3 of version 2 was deleted.

In summary, $RBVM$ provides a good basis for supporting an assortment of different types of basic queries on versioned XML documents. However, the rich framework of XML allows the user to express more complex types of queries that we have not addressed here. Issues of particular interest are:

- *Content-based selection.* In many queries, this is also combined with structural projections. Selection is easily performed on versions (or their structural projections) produced by the algorithms previously discussed. Further optimization is also possible by pushing selection onto the referenced elements of the previous version.
- *Structural queries* including the filter queries and the path expression queries found in XQuery [33].
- *Indexing.* Various index structures can be used to expedite the execution of queries discussed in this section. In particular, a sparse index can be used to search for an element, given a version number and the preorder number of the element in that version. Such an index can be implemented as a standard B$^+$-tree, and can expedite the execution of various queries.

The problem of supporting complex queries on XML documents presents interesting research challenges even when support for multiple versions is not required. While we believe that techniques similar to those used for structural projection can be useful in addressing these problems, the main issues remain unsolved and pose an interesting topic for future research.

4.3 Transport level: RBVM in XML

The reference-based scheme allows a very natural XML-based representation of XML documents at the external level, i.e., at the level where documents are viewed through browsers or exchanged between web sites (transport level). This is because the $RBVM$ scheme preserves the structure of the original document, and in fact the whole document history can be naturally viewed as yet another XML document. In fact, we will next show how to derive a DTD for the version repository (i.e., the document history) from the DTD of the original document. Thus, all the $RBVM$ repository can be represented in XML and seamlessly viewed, retrieved, transported, or restructured by applications which understand the DTD.

Since $RBVM$ itself has an ordered-tree structure, it can be naturally described by DTD. The DTD of a $RDVM$ repository can be derived from the original DTD of documents simply by the following step :

- Three new DTD elements are defined to represent: (i) the repository, (ii) the versions; and (iii) the reference records;
- For each element defined in the original DTD (except the root) its content model is modified to include a reference record element as an alternate;
- An ID attribute is added to each element (that does not have one already).

Figure 10 shows a DTD, simplified from the SIGMOD Record page and its version DTD. The version DTD starts with the definitions of the root element $Repository$ whose content

```
materialize(K, S, E, dx, dy)
        {split(PL(K), S, E, L1, L2, L3);
         for each (I:J, N, L) in PL(K)
                 out(K):= out(K)||produce1(K,I,J, N-I,L))
         Temp:= produce1(K, S, E, dx, dy);
         out(K):= out(K)||extract(L2, Temp);
         PL(K):= L3;
         return(Temp)}
```

Fig. 9. The structural projection algorithm

```
     <!-- ORIGINAL DTD -->
G.1  <!ELEMENT OrdinaryIssuePage
       (volume_info,sectionList)>
G.2  <!ELEMENT volume_info (#PCDATA)>
G.3  <!ELEMENT sectionList(sectionListTuple)*>
G.4  <!ELEMENT sectionListTuple(sectionName,articles)>
     ...
     <!-- VERSION DTD  -->
N.1  <!ELEMENT Repository (Version)+> N.2  <!ELEMENT Version
(OrdinaryIssuePage)>
     <!ATTLIST Version v_number CDATA #REQUIRED>
N.3  <!ELEMENT RefRecordPair>
     <!ATTLIST RefRecordPair v_number CDATA>
     <!ATTLIST RefRecordPair start_element IDREF>
     <!ATTLIST RefRecordPair end_element IDREF>
N.4  <!ELEMENT OrdinaryIssuePage
       (volume_info,sectionList)>
N.5  <!ELEMENT volume_info ((#PCDATA)|RefRecordPair)>
     <!ATTLIST volumn_info id ID>
N.6  <!ELEMENT sectionList
       ((sectionListTuple)* | RefRecordPair)>
     <!ATTLIST sectionList id ID>
N.7  <!ELEMENT sectionListTuple
       ((sectionName,articles) | RefRecordPair)>
     <!ATTLIST sectionListTuplle id ID>
     ...
```

Fig. 10. A sample DTD and its version DTD

is a list of version elements. Then the version element is defined at point N.2, which contains one occurrence of the root element of the original DTD, $OrdinaryIssuePage$, which is defined at G.1. The version type also has an attribute to keep its version number. The reference record is defined at point N.3, which is an empty element with three attributes that represent: (i) the referenced version number; (ii) the id of the starting element; and (iii) the id of the end element of the segment to be copied from the specified version. The definition of the root element of the original DTD, (i.e., $OrdinaryIssuePage$) remains unchanged; however, the content model of every other element is extended to include the $RefRecordPair$ element as an alternate, which stores the information $S : E$ where S and E are the respective start and end of the segment being copied from the previous version. For example, the $sectionList$ element defined at G.3 of the original DTD is changed to include a $RefRecordPair$ element as an alternate content. Based on the above procedure, the version DTD can be automatically derived from the original DTD and be used by any application which understands the content of the repository.

Recently, a schema definition language, called XML Schema [21], was proposed to support richer semantics for XML documents. The previous procedure to derive a version DTD from the DTD of the original document can be easily extended to derive the version XML Schema from the XML Schema of the original document [8].

4.4 Revised usefulness-based management

As versions progress, old pages contain fewer and fewer elements that are still valid for the current version, and materializing such a version requires accessing an increasing number of pages. To solve this problem, we want to introduce a usefulness-based management scheme similar to that used for the edit script method. Here, however, if we salvage the valid elements in a page, we have also to salvage all the pages containing references pointing to the old pages (otherwise our materialization algorithms will still retrieve the old pages). In a nutshell, we need a more sophisticated and global storage management policy, described next.

Let us start by introducing the notion of *usefulness of the support set* of a data page p filled with entries (reference records and actual objects) of a Version k. Let $S_k(p)$ denote the logical segment of Version k described by p; obviously, $S_k(p)$ can be materialized by taking the actual data entries in p and recursively expanding its references into the logical segments they represent. The pages $p, p_1, p_2, \cdots, p_n$ that must be accessed to materialize p will be called the *support set for* p. Then we have the following definition.

```
----------------------------VERSION 1------------------
p1:  (A,1,0), (B,2,1), (C,3,1), (D,4,1),
p2:  (E,5,2), (F,6,2),(G,7,2), (H,8,1),
p3:  (I,9,1), (J,10,1), (K,11,1), (L,12,2),
p4:  (M,13,2), (N,14,1), (O,15,1), (P,16,1).
---------------------------- VERSION 2------------------
p5:  (1:4,1,0), (Q,5,2), (R,6,2), (5:8,7,0),
p6:  (S,11,1),(9:11,12,0), (13:14, 15,0), (16:16,17,0).
----------------------------VERSION 3------------------
p7:  (A,1,0), (B,2,1), (D,3,1), (E,4,2),
p8:  (F,5,2), (G,6,2),(H,7,1), (J,8,1),
p9:  (K,9,1), (M,10,2), (T,11,1).
```

Fig. 11. The $RBVM$ scheme with usefulness-based management

```
INSERT(V)
 { for (each element E in V)
    { 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;
       }
   }
```

Fig. 12. Adding a new version in the $RBVM$ scheme

Definition: The usefulness of p's support set, denoted by $US(p)$, is defined as the ratio of the size of $S(p)$ over the total size of pages $p, p_1, p_2, \cdots, p_n$:

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

where σ is the size of a secondary storage page. □

Example. For Version 1 in Fig. 7 we have $US(p1) = US(p2) = US(p3) = US(p4)$. For Version 2 in the same figure, we have that $p5$'s segment consists of 10 elements: $S(p5) = [\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{Q}, \mathtt{R}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}]$. The support set of $p5$ is $p1, p2, p5$; thus, $US(p5) = (10 \times b)/(12 \times b) = 10/12$. $S(p6)$ consists of 7 elements, while its support set is $p3, p4, p6$; thus, $US(p6) = 7/12$.

For Version 3, we find that $S(p7) = [\mathtt{A}, \mathtt{B}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{J}, \mathtt{K}, \mathtt{M}]$; to materialize this segment, we need to access all the previous pages: $p1, p2, p3, p4, p5, p6, p7$; thus $UP(p7) = 10/28$. Page $p8$ contains only one entry; this page is not filled completely since it is the last page of the last version. By convention, if p is the last page of the current version and this page is not completely filled yet, we set $US(p) = 1$; thus $US(p8) = 1$ in our example.

To guarantee low I/O cost, we define again a minimum required usefulness U_{min}, but with a revised page-usefulness definition. A page p will be called $useful$ as long as $US(p) \geq U_{min}$. The next question is what to do with pages whose usefulness is below U_{min}. For example, assume that we set $U_{min} = 50\%$. Then we see that every page in Version 1 and 2 of our example is above the threshold. However, page $p7$ of Version 3 does not satisfy the requirement. Thus, we will not store the current content of page $p7$ shown in Fig. 7: Instead, the page is *expanded to its actual content*, by storing $S(p7) = [\mathtt{A}, \mathtt{B}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{J}, \mathtt{K}, \mathtt{M}]$ starting with the current page $p7$, and continuing with the two pages that follow. The rest of Version 3, i.e., $(\mathtt{T}, 11, 1)$, is stored after that.

With this usefulness revision, the $RBVM$ scheme of Fig. 7 is actually stored as in Fig. 11. Now $US(p7) = US(p8)$. We also set $US(p9) = 1$ since this page is not filled yet. Eventually, page $p9$ will be filled with entries from a new Version 4; at this point, the usefulness of $US(p9)$ is recomputed and if below the threshold the page will be expanded to its actual content. As illustrated by this example, the usefulness policy for $RBVM$ often copies segments of larger granularity than the edit-based usefulness policy. The complete version insertion algorithm can thus be summarized as follows:

Two further refinements are used in the version insertion algorithm to minimize unnecessary copying. To materialize page 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 can stop the materialization as soon as the first page boundary is reached, and return to step 2; the second page might in fact still be useful, and similar considerations hold for the pages that follow. Let us use the example in Fig. 11 where the usefulness of page p7 is below U_{min} so its logical segment must be copied. Instead of materializing the whole $S(p7)$, the first four elements are first copied and stored into $p7$. Then, the remaining logical segments (8:10,6,0),(13:15,8,0), and (T,11,1) are stored in the next new page, $p8$. If page $p8$ is useful, then there is no need to copy the rest of the elements. In this case, page $p8$ is still useless, so the copying continues till all data pages are useful or the end of Version 3 is met.

4.4.1 Complexity of the $RBVM$ scheme

• *Version retrieval I/O cost.* The cost for reconstructing any given version can be computed as the number of pages that have to be read into main memory. We next show that the reconstruction of a version V of size $size(V)$ needs $\frac{1}{U_{min}} \times \frac{size(V)}{\sigma}$ pages, where σ denotes the size of a page.

Let us assume that the reference-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 respective logical segments for these pages. Since by construction all these pages are useful, the cost of materializing a segment, S_i, is bound by $\frac{size(S_i)}{U_{min}}$. 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: $size(V)/U_{min}$.

• *Storage cost.* In fact, the reference-based scheme consists of three kinds of objects: actual objects, copied objects 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 S_{chg}. With a similar argument as for UBCC, the total number of copied objects is bound by:

$$S_{chg} \times \frac{U_{min}}{1-U_{min}}$$

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. Combining these three parts, the storage of the reference-based scheme is $S_{chg} \times (K + \frac{1}{1-U_{min}})$. That is, the $RBVM$ storage cost is linear with the total size of changes by a constant determined by the required minimum usefulness U_{min} and the ratio K. The linear cost of the $RBVM$ scheme is also confirmed by the experimental performance evaluation discussed in the next section.

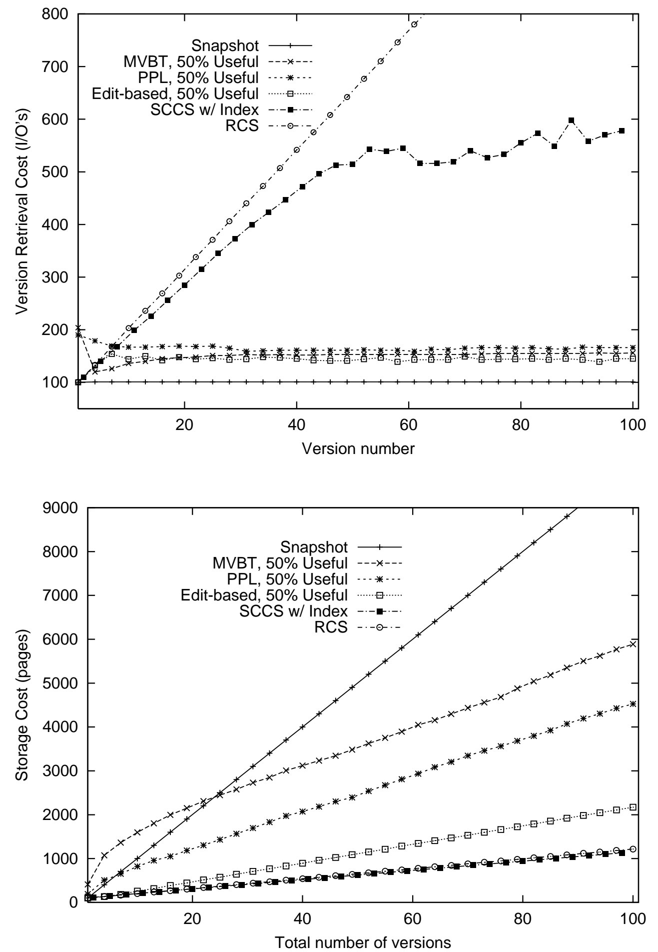


Fig. 13. Version retrieval and storage cost comparison

5 Performance of the edit-based scheme

We first evaluate the edit-scheme (UBCC) against alternative schemes proposed in the literature, including:

- The SCCS scheme that has recently been used to archive scientific data [1];
- The "snapshot" approach that simply keeps a copy of each document version;
- The RCS scheme;
- The Multiversion B$^+$-tree [28];
- The Partially-persistent list method [31].

The Multiversion B$^+$-tree (MVBT) and the Partially-persistent list (PPL) methods are discussed in the appendix. Both schemes are based on the idea of extending the basic data structure (namely the B$^+$-tree and the list) by making them partially persistent[2]. The B$^+$-trees and lists preserve the document preorder for each version of the document. Both schemes use a form of page usefulness that is similar to the edit-based UBCC (i.e., the usefulness of a page is based on the number of valid records it contains). Thus, each page is assigned a (preorder) range and is considered useful for as long as it contains a minimum number of valid objects from this range. The SCCS scheme is implemented by storing successive versions of a given object sequentially. Note that this may incur high update cost when inserting a new version; this problem can be solved, at the cost of some additional storage, by not filling newly created pages completely. In addition, for the SCCS scheme we facilitate a sparse index which for each version identifies all pages with objects valid in this version.

We use a document evolution with the following characteristics: (i) the size of each version is approximately 100 pages; (ii) each version changes about 20% from the previous version (half of the changes are insertions and the other half are deletions); (iii) changes are uniformly and randomly distributed among data pages; and (iv) the document evolution had a total of 100 versions. The page size is set to 4,096 bytes.

Figure 13 shows the version retrieval cost (the number of page I/O's needed to reconstruct a version) and the storage cost (number of pages). In this figure, the minimum usefulness parameter for UBCC, MVBT and PPL was 50%, i.e., $U_{min} = 0.5$. The snapshot scheme clearly has the minimal version retrieval cost, since each version is already stored in its entirety on disk. The RCS method needs to read the file until the target version. Therefore, retrieving later versions gets more expensive. The version retrieval cost of the SCCS scheme is also high since when a page is accessed it may provide very few valid objects for the requested version (at worst, a single object). The use of the index in SCCS improves the

[2] A data structure is *persistent* if it maintains its past; it is partially persistent if only its latest version can be updated [10]

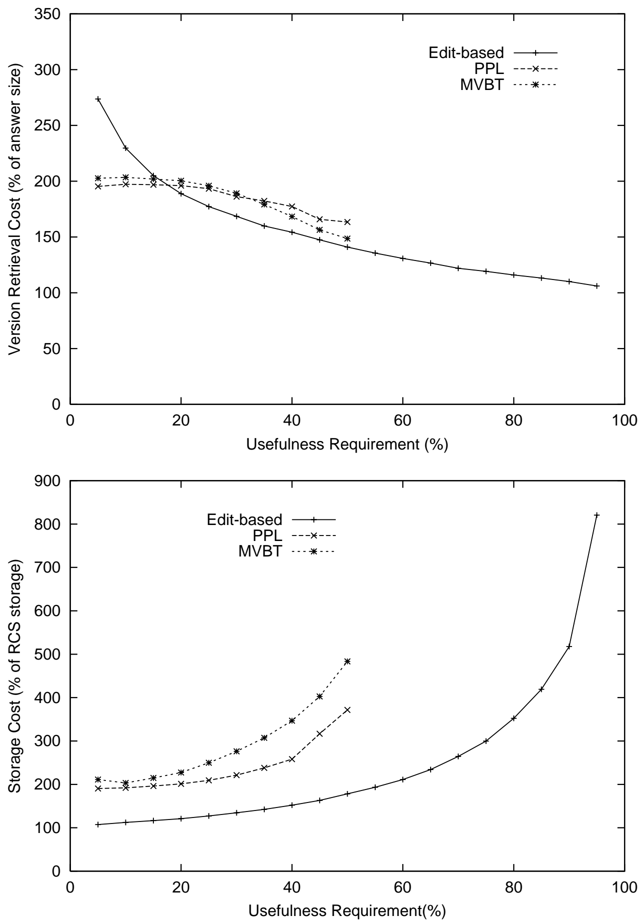


Fig. 14. Version retrieval and storage cost vs usefulness

method's retrieval cost when compared to RCS. The schemes that use page usefulness (UBCC,MVBT,PPL) have version retrieval cost that is proportional to the size of the reconstructed version (for all three schemes the retrieval cost appears approximately parallel to the horizontal axis since in this experiment, the average version size remains the same, about 100 pages). The overhead against the snapshot scheme is because useful pages may include some non-valid objects for the retrieved version. Among the three approaches, the edit-based (UBCC) scheme uses less I/O since the answer is clustered in fewer pages. Pages in the MVBT and PPL approaches may contain empty space since they can only store objects in their range. The MVBT is slightly better than the PPL because PPL uses some page capacity for the SA arrays.

While the snapshot scheme provided the minimal retrieval cost, its storage cost is too expensive (at worst it is quadratic). RCS and SCCS have minimal storage cost since conceptually both store only the version change information. The space of SCCS is slightly smaller because RCS explicitly stores delete records. The space of all the page-usefulness schemes grows linearly with the number of changes (which increases with the number of versions). In particular, the partially persistent schemes (PPL, MVBT) use more space than UBCC where besides deletions, insertions resulting in a page split, also cause the copying of the old elements into a new page. By contrast, in the UBCC scheme, copies of old elements can only be the results of deletions that lower the usefulness of the page below the threshold. 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 for UBCC, MVBT and PPL, we run the same experiment as above, but for different U_{min}. The experimental results are illustrated in Fig. 14. The performance of the PPL and MVBT schemes is depicted until U_{min} = 50% since this is the highest usefulness they can achieve. In contrast, the UBCC scheme can be used with any $U_{min} \leq 100\%$. 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. Another interesting observation has to do with the behavior at very small U_{min}. 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_{min} 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. 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 14 depicts the storage cost as a percentage over the (minimal) RCS storage. Higher U_{min} 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 Fig. 14, 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. Minimum usefulness is again set to 50%. The results are shown in Fig. 15. All page-usefulness schemes have very close version retrieval performance that is proportional to the version size. The edit-based storage cost is very close to the minimal storage of RCS and SCCS, because the small deletion percentage rarely causes UBCC to copy useless pages. The MVBT and

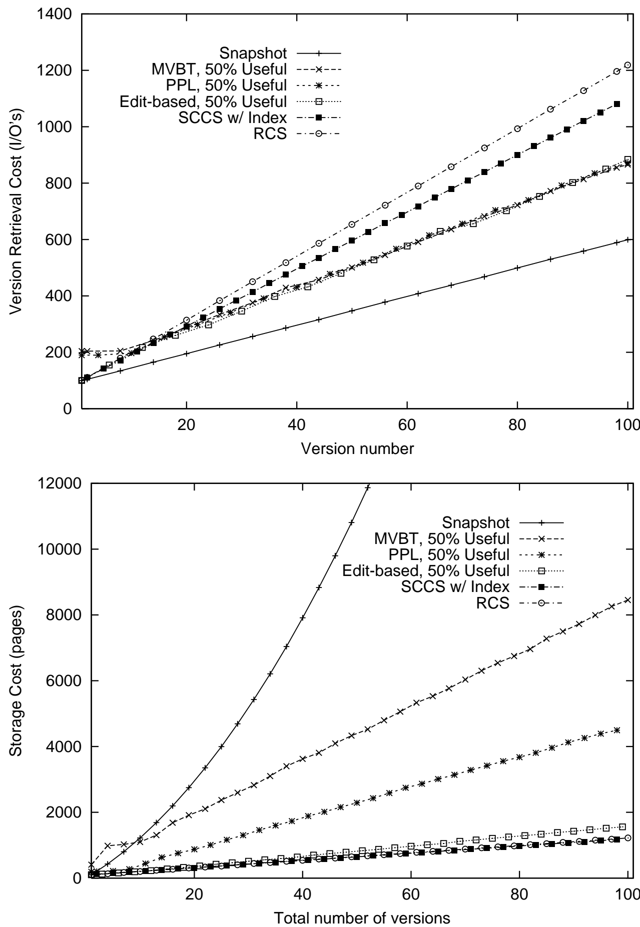


Fig. 15. Version retrieval and storage cost with increasing document size

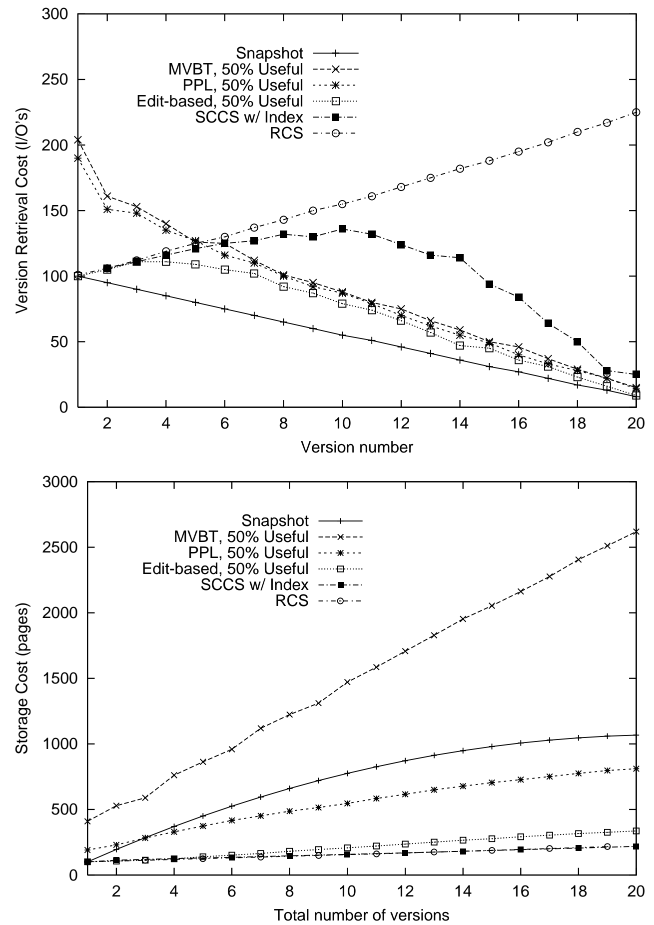


Fig. 16. Version retrieval and storage cost with decreasing document size

PPL both use more space than UBCC because insertions trigger more copies. Observe that while the storage required by the snapshot scheme grows quadratic with the version number, the other schemes still use linear storage.

Figure 16 depicts the performance for a document that decreases in size as it evolves. Here the document size decreases by 5% per version, as the result of 5% insertions and 10% deletions. The version retrieval cost of all page-usefulness schemes decreases as the version size decreases. Similarly, SCCS retrieval cost eventually decreases since later versions have fewer elements and thus less pages are accessed by the SCCS index. 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 of RCS and SCCS.

From the above results, we conclude that page usefulness provides version retrieval cost that is proportional to the size of the target version at the expense of some extra (linear) space. Setting the minimum usefulness enables performance tuning. Furthermore, the edit-based (UBCC) scheme is more robust than the partially-persistent methods (MVBT, PPL) since it consumes less space and has better retrieval performance.

6 Performance of reference-based scheme

We now proceed with an evaluation of the reference-based scheme. Since we determined that the edit-based scheme is the most efficient among those discussed in the last section, we will now compare the reference-based approach against it.

Figure 17 shows the comparison of the two methods using the original document evolution dataset. We depict also RCS and snapshot to serve as the baseline cases. The page usefulness definition of the reference-based scheme considers the collective usefulness of the whole segment each page contains. When the usefulness of a segment falls below the threshold, several pages are normally involved; this leads to better clustering. On the other hand, pages within a segment can compensate each other on their usefulness; hence some segment pages may have fewer valid objects. This larger granularity causes its retrieval cost to fluctuate around the smoother line of the edit-based scheme. Some valleys in the reference-based curve approach the performance of the snapshot case. Its peaks exceed the 150 page level of the edit-based scheme but remain well below the theoretical worst case of 200 pages ($U_{min} = 0.5\%$). Nevertheless, on average, the reference-based and edit-based schemes have similar retrieval performance. The storage cost of the two methods is basically identical.

Figures 18 and 19 show the performance results for an increasing and a decreasing document evolution. Again the two

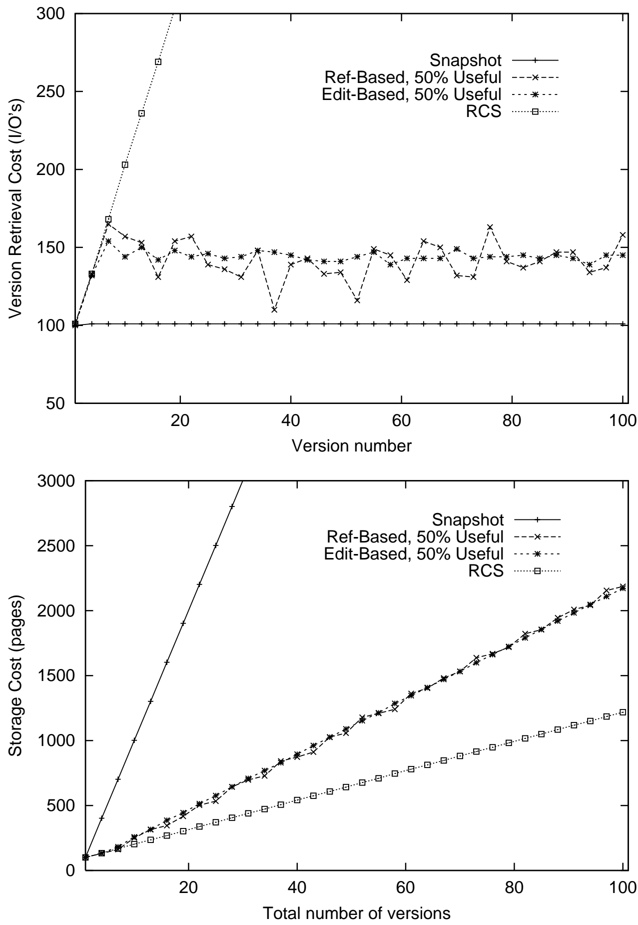


Fig. 17. Version retrieval and storage cost with 50% usefulness

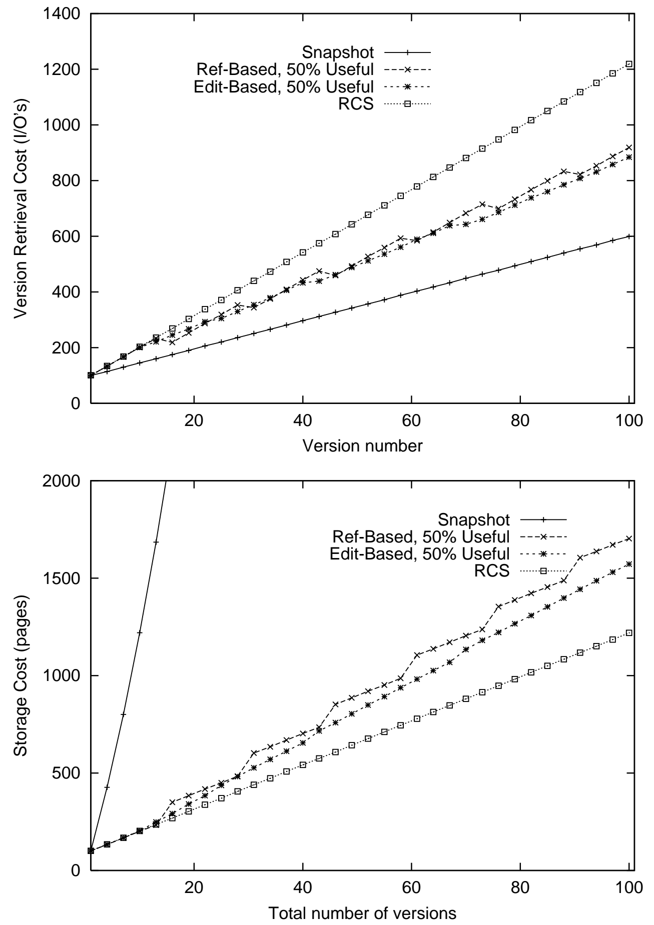


Fig. 18. Version retrieval and storage cost with increasing document size

schemes perform quite similarly. The reference-based scheme requires slightly more storage in the increasing document evolution experiment (Fig. 18) since when a segment is materialized, it corresponds to a larger document portion.

6.1 Indirect versus direct reference records

The reference records used in RBVM are $indirect$ in the sense that they always refer to the previous version, whether the referred elements are actually stored there or in an earlier version. As previously discussed, when materializing a reference record, the sequence of previous versions are recursively visited to locate the actual elements. The referencing cost is controlled by the fact that we apply the usefulness-base copy technique whenever the cost of materializing a page gets over a certain threshold.

An alternative scheme which can reduce the referencing cost is using *direct reference records*. Instead of always referring to the previous version, direct reference records directly refer to the version where the actual elements are located. For example, in Fig. 6 the reference record (v2.1) is an indirect record where the actual element are physically store in Version 1. Using a direct reference record, the record will be represented as (v1.1) which directly points to the actual element in Version 1. Therefore, when materializing Version 3, its first chapter is directly retrieved from Version 1 without passing Version 2. However, there are also major drawbacks with the direct referencing scheme.

The first problem is that the fragmentation of the document requires more reference records for the direct scheme than the indirect one. For example, assume that the original version consists of eleven elements:

$$(E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8, E_9, E_{10}, E_{11})$$

and the second version is derived by deleting E_4 and E_8. Then, the RBVM representation with indirect reference records is

$$(V_1, (1, 3)), (V_1, (5, 7)), (V_1, (9, 11))$$

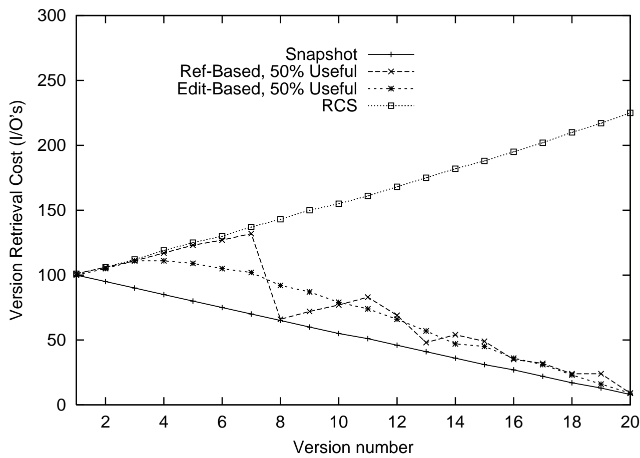
and its RBVM representation with direct reference scheme is the same. Let the third version be generated from Version 2 by deleting element E_2 and E_6. Then the indirect-referencing RBVM of Version 3 is

$$(V2, (1, 1)), (V2, (4, 4)), (V2, 6, 9)$$

while its direct-referencing RBVM representation is

$$(V1, (1, 1)), (V1, (5, 5)), (V1, 7, 7), (V1, 9, 11)$$

Typically, therefore, more records are created by the direct referencing scheme. To better understand this effect, we have performed three sets of experiments, to determine the total



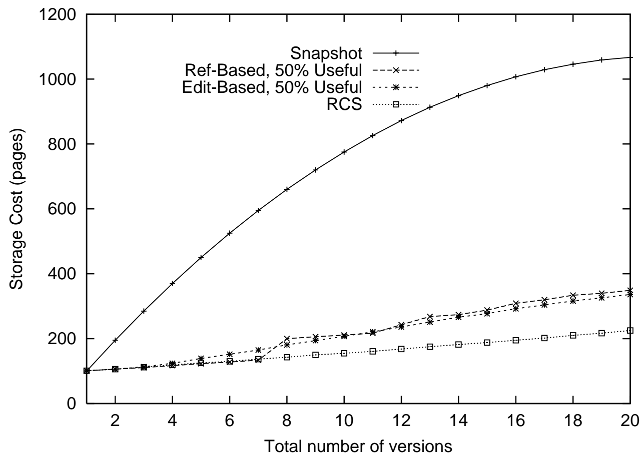


Fig. 19. Version retrieval and storage cost with decreasing document size

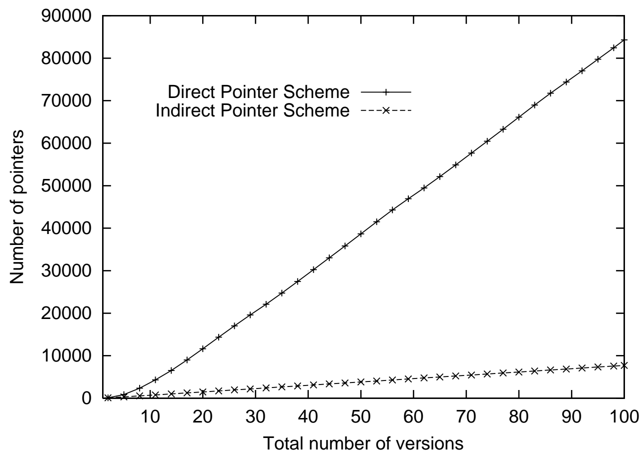


Fig. 20. Reference count growth with 10% insertions and 10% deletions

number of reference records generated by the indirect referencing scheme and the direct referencing scheme. Figures 20, 21, and 22 show the results of these experiments. Figure 20 shows the result of an evolution, where each new version contains 10% insertions and 10% deletions randomly distributed at various points of the document. As shown, the total number of reference records generated by the direct referencing scheme is significantly larger than in the indirect referencing

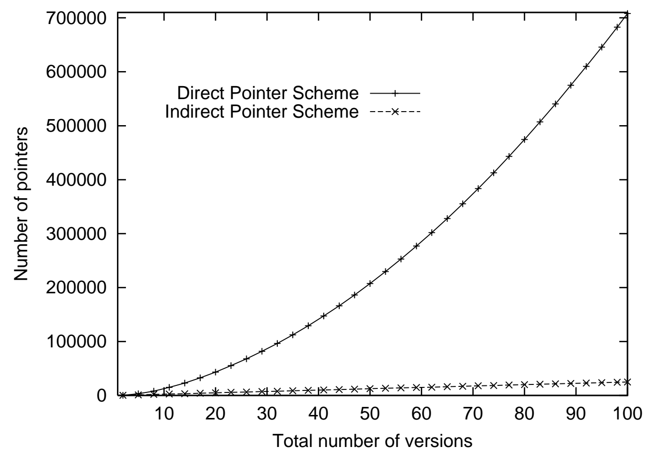


Fig. 21. Reference count growth with 10% insertions and 5% deletions

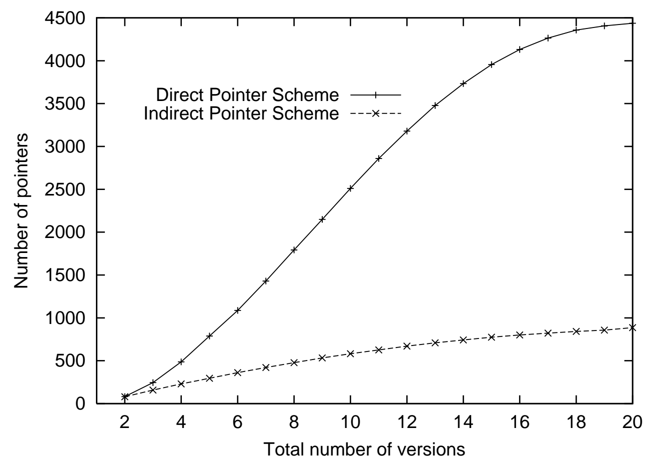


Fig. 22. Reference count growth with 5% insertions and 10% deletions

scheme. Similar results are obtained for an evolution with increasing document size as shown in Fig. 21 and an evolution with decreasing document size as shown in Fig. 22. Therefore, the indirect reference scheme takes less storage than the direct reference one while the I/O cost of its materialization is controlled by page usefulness.

7 Conclusion

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

For the physical level, we proposed a temporal clustering technique based on the notion of 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 proposed the *edit-based* scheme that improves on RCS by reducing the average version

retrieval cost, at the cost of a small (linear) storage overhead. Furthermore, our scheme separates the edit scripts from the document objects to assure faster retrieval.

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 scheme; in fact, by using preorder node numbers, 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 (single version) document. This representation is very suitable for encoding the whole document history as yet another XML document that can be viewed on a browser, or exchanged between different sites (transport level). The representation is also conducive to expressing and processing efficiently content and evolution queries on multiversion documents. Furthermore, we have extended the page-usefulness clustering technique to this representation as well.

A thorough experimental study was presented where the edit-based schemes was compared against the original RCS and SCCS schemes, the snapshot scheme (that simply stores each version in its entirety), the Multiversion B$^+$-tree, and the Partially-Persistent List method. We showed that the edit-based scheme performed the best among these; so we then compared the reference-based scheme against the edit-based one and showed that it has the same average performance, with somewhat larger variations between versions.

A main conclusion that follows from this work is that the management of multiversion XML documents presents many research opportunities, and requires new techniques that are often closer to temporal databases than traditional versioning schemes (although the need to preserve the document order and support its complex internal structure makes the problem quite different from that of traditional transaction-time databases). In particular, while the techniques presented here have addressed successfully many issues, they have left many unresolved. For instance at the storage level, there is the important issue of the role that compression can play in improving the storage efficiency of the archive [1]. Another important topic left for future research is that of efficient support for complex queries, including those containing structural joins (i.e., path expressions). These complex queries provide a research challenge even when multiple versions are not present [33,39,40]. For multiversion documents this challenge is compounded by the fact that structural queries must be supported in combination with queries on the evolution of documents.

The temporal aspects of these queries also pose interesting problems in terms of language constructs that need to be supported for expressing them. For instance, it is far from clear that languages such as XQuery [33] can express queries similar to those expressible in languages such as TSQL2 [35] on transaction-time databases.

Acknowledgements. The authors would like to thank the referees for many suggested improvements, and Stott Parker for bringing the issue of indirect versus direct references into focus. This research was partially supported by NSF grants IIS 0070135, IIS-9907477, EIA-9983445, and the Department of Defense.

A Appendix: the partially persistent approaches

An alternative way to maintain a multiversion document is to utilize a data structure that preserves the document's order (for example a B$^+$-tree or an ordered list). Document versioning is then reduced to making this data structure partially persistent [10]. We examine two approaches: (i) using a multiversion B$^+$-tree; and (ii) using a partially-persistent list.

A.1 Utilizing a multiversion B$^+$-tree

Consider a B$^+$-tree whose leaves contain records with keys 1, 2, 3,. . ., where record l stores the object in the $l^{\mathtt{th}}$ position in the document preorder. Since each object insertion/deletion affects the preorder 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 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 encode 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, let the first object in version V_1 be 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.

There have been various approaches to making a B$^+$-tree partially persistent [26,28,29]. In our experiments we used the MVBT [28] since its code was readily available to us. The MVBT 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.

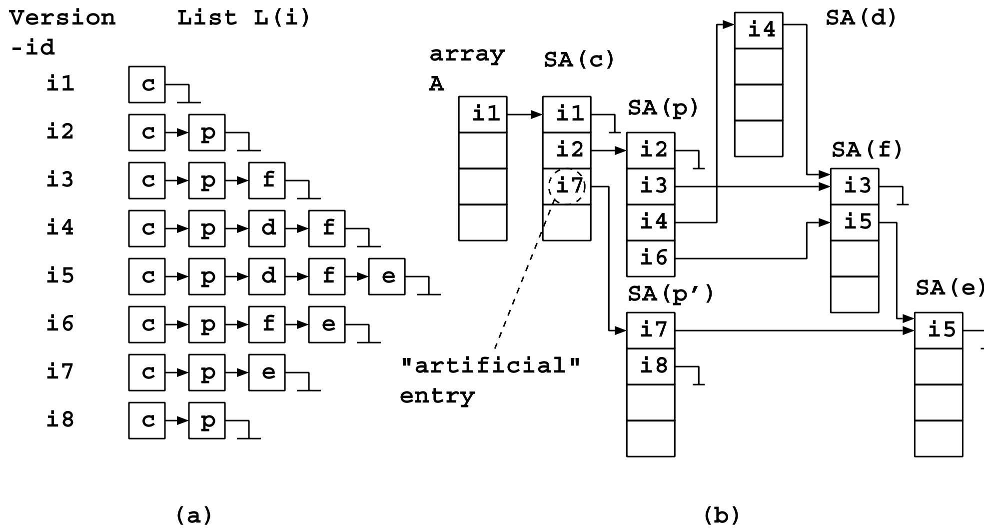


Fig. 23. A partially persistent list example

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 [28].

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 [26,27]. 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 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 detailed discussion of the page split algorithm can be found in [28].

The space used by the above splitting/merging policies is still linear in 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.

A.2 Utilizing a partially persistent list

Various notions of partially persistent lists have appeared in the temporal database literature. Our discussion follows the approach outlined in [31] on how to make an ordered list partially persistent. In [30] a scheme to support non-ordered partially persistent lists is presented. [29] 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 [28].

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 Fig. 23.

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.

References

1. P. Buneman, S. Khanna, K. Tajima, W.C. Tan (2002) Archiving scientific data. SIGMOD 1–12
2. D. Beech, B. Mahbod (1988) Generalized version control in an object-oriented database. IEEE 4th International Conference on Data Engineering, Feb
3. F.W. Burton, J.G. Kollias, D.G. Matsakis, V.G. Kollias (1990) Implementation of overlapping B-trees for time and space efficient representation of collections of similar files. Comp J 33(3):279–280
4. M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita (1986) Object and file management in the EXODUS extensible database system. In: Proc. VLDB Conference, pp 91–100
5. S. Chawathe, H. Garcia-Molina (1998) Representing and querying changes in semistructured data. Proc. International Conference on Data Engineering, pp 4–13.
6. S. Chawathe, S. Abiteboul, J. Widom (1999) Managing historical semistructured data. TAPOS 5(3):143–162
7. S.-Y. Chien, V.J. Tsotras, C. Zaniolo (2000) Version management of XML documents. WebDB 2000 Workshop, Dallas, Tex., USA
8. S.-Y. Chien, V.J. Tsotras, C. Zaniolo (2001) Efficient management of multiversion documents by object referencing. UCLA Tech. Rep. No 010024, June
9. D.E. Knuth (1998) The art of computer programming, vol. 1: fundamental algorithms. Addison-Wesley, Reading, Mass., USA
10. J.R. Driscoll, N. Sarnak, D. Sleator, R.E. Tarjan (1989) Making data structures persistent. J Comput Syst Sci 38:86–124
11. M.C. Easton (1986) Key-sequence data sets on inedible storage. IBM J Res Dev 30(3):230–241
12. R.H. Katz, E. Change (1987) Managing change in computer-aided design databases. Proc. VLDB Conf., Brighton, UK, Sep
13. D. B. Leblang (1994) The CM challenge: configuration management that works. In: W.F. Tichy (ed) Configuration management. Wiley, New York, pp 1–38
14. A. Marian, S. Abiteboul, G. Cobena, L. Mignet (2001) Change-centric management of versions in an XML warehouse In: Proc. 27th VLDB, Rome, Italy, Sept
15. G. Ozsoyoglu, R.T. Snodgrass (1995) Temporal and real-time databases: a survey. IEEE Trans Knowl Data Eng 7(4):513–532
16. W.F. Tichy (1985) RCS – A system for version control. Software Pract Exp 15(7):637–654
17. V.J. Tsotras, N. Kangelaris (1995) The snapshot index, an I/O-optimal access method for timeslice queries. Inf Syst 20(3):237–260
18. P.J. Varman, R.M. Verma (1997) An efficient multiversion access structure. IEEE Trans Knowl Data Eng 9(3):391–409
19. World Wide Web Consortium (1999) XML Path Language (XPath) Version 1.0. Nov. 16. See http: //www. w3. org/ TR/ xpath.html
20. WWW Distributed Authoring and Versioning (webdav) (2002) See: http://www. ietf. org/ html. charters/ webdav-charter. html
21. XML Schema (2002) World Wide Web Consortium. See: http://www. w3. org/ XML/ Schema
22. S.J. Yoo, P.B. Berra, Y.K. Lee, K. Yoon (1996) Version management in structured document retrieval system. Proc. 8th Int. Conf. Software Engineering and Knowledge Engineering, pp 537–544
23. K. Zhang (1995) Algorithms for the constrained editing distance between ordered labeled trees and related problems. Pattern Recognition 28(3):463–474
24. M.J. Rochkind (1975) The source code control system. IEEE Trans Software Eng SE-1:364–370
25. G. Cobena, S. Abiteboul, A. Marian (2002) XyDiff Tools detecting changes in XML documents. http://www-rocq.inria.fr/∼cobena
26. D. Lomet, B. Salzberg (1989) Access methods for multiversion data. Proc. SIGMOD. pp 315–324
27. V.J. Tsotras, N. Kangelaris (1995) The Snapshot Index: an I/O-Optimal access method for timeslice queries. Inf Syst 20(3):237–260
28. B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer (1996) An asymptotically optimal multiversion B-tree. VLDB J 5(4):264–275
29. P. Varman, R. Verma (1997) An efficient multiversion access structure. IEEE TKDE 9(3):391–409
30. G. Kollios, V.J. Tsotras (2002) Hashing methods for temporal data. IEEE TKDE (to appear)
31. A. Kumar, V.J. Tsotras, C. Faloutsos (1998) Access methods for bi-temporal databases. IEEE Trans Knowl Data Eng 10(1):1–20
32. I. Tatarinov, Z.G. Ives, et. al (2001) Updating XML. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, pp 413–424
33. XQuery 1.0 (2002) An XML query language . http://www.w3.org/TR/xquery/
34. J.D. Ullman (1988) Principles of database and knowledge-base systems, vol. 1. Computer Science, New York
35. C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, R. Zicari (1997) Advanced database systems. Morgan Kaufmann, San Francisco
36. National Archives of Australia's Policy Statement Archiving Web Resources (2002) A policy for keeping records of web-based activity in the commonwealth government. http://www.naa.gov.au/recordkeeping
37. B. Kahle, J. Alexa, et al (2002) The Internet archive – the wayback machine – surf the Web as it was. http://www.archive.org/index.html
38. M.A. Noronha, L.G. Golendziner, C.S.D. Santos (1998) Extending a structured document model with version control. Proc. Int. Database Engineering & Applications Symposium, pp 234–243
39. D. Srivastava, S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, Y. Wu (2002) Structural joins: a primitive for efficient XML query pattern matching.. In: Proc. 18th International Conference on Data Engineering (ICDE 2002), pp 141–152, IEEE Computer, New York

40. Q. Li, B. Moon (2001) Indexing and querying XML data for regular path expressions. In: Proc. VLDB 2001, Roma, Italy, September, pp 361–370
41. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. J. DeWitt, J. F. Naughton (1999) Relational databases for querying XML documents: limitations and opportunities. In: Proc. VLDB, pp 302–314
42. D. Florescu, D. Kossmann (1999) Storing and querying XML data using an RDBMS. Data Eng Bull 22(3)
43. S.-Y. Chien, V.J. Tsotras, C. Zaniolo, D. Zhang (2002) Efficient complex query support for multiversion XML documents. In: Proc. 8th Int. Conference on Extending Database Technology (EDBT 2002), Prague, Czech Republic, pp 161–178
44. S.-Y. Chien, V.J. Tsotras, C. Zaniolo (2001) Efficient management of multiversion documents by object referencing. In: Proc. VLDB'01, Roma, Italy, Sept