Efficient Management of Multiversion Documents by Object Referencing

Shu-Yao Chien
CS Dept. UCLA
csy@cs.ucla.edu

Vassilis J. Tsotras*
CSE Dept., UC Riverside
tsotras@cs.ucr.edu

Carlo Zaniolo*
CS Dept. UCLA
zaniolo@cs.ucla.edu

Abstract

Traditional approaches to versioning documents are edit-based, and represent successive versions using edit scripts. This paper proposes a reference-based versioning scheme that preserves the rich logical structure of the evolving document via object references. This approach produces better support for queries, and reconciles the storage-level and transport-level representations of multiversioned XML documents. In particular, we present efficient algorithms for supporting projection and selection queries, and for querying the document evolution history. Then, we show that our representation is also efficient at the transport level, where XML documents are exchanged between remote parties. In fact, with the reference-based scheme, an XML document's history can also be viewed and processed as yet another XML document. Finally, we demonstrate the effectiveness of the new scheme at the storage level, for which we define a usefulness-based page management policy, adapted from transaction-time databases, to ensure efficient temporal clustering between versions. The experimental evaluation of the new scheme against previous representations used in temporal databases and persistent-object managers shows the performance advantages of the new approach.

1 Introduction

With the advent of the WWW, an abundance of semistructured documents is stored and disseminated from different application domains. Since such documents evolve (updates, new releases) the problem of document versioning has become of significant interest for content providers, cooperative work, and information systems in general. Various stardarization groups, have recognized the importance of versions [21], but left it for later standards, because of the many research issues still unresolved. This situation creates an exciting window of research opportunities. While there has been very little previous work on versioning web based semistructured documents [5, 21], much relevant work was done for other applications, such as software configuration management [11], CAD systems[10] and temporal databases [1, 3, 9, 12, 16, 18, 19].

Some of the problems occurring in multiversion documents are similar to those of transaction-time databases, where object histories are maintained (new objects are added without discarding the old ones) [14]. Using timestamping, various efficient indexing and clustering techniques have been proposed for temporal relations [1, 3, 9, 12, 16, 18, 19]. Version management schemes have also been proposed in OODBs; however, they are not designed to support documents and to optimize the retrieval of complex documents [2, 4, 8, 10].

Change management for semistructured data has been proposed in Chorel [5]; the focus though is on modeling version changes for individual objects, and the performance of storing and retrieving complete document versions was not discussed.

Two version management schemes used in software configuration management [11] are RCS [17] and SCCS [15]. RCS uses an *edit-based* approach for representing multiple versions of an evolving textual object. A recent approach to XML versioning based on RCS is [13]. Typically, RCS [17] stores the most current version verbatim, while previous revisions are represented via *reverse edit scripts*. These scripts describe how to go backward in the object's development history. For any version, but the current one, extra processing is needed to apply the reverse edit script to generate the old version. The symmetric approach to the problem uses instead a forward edit script where the original versions are stored intact, and successive versions are generated by following the script. A timestamp-based scheme is instead used for SCCS [15], where each textual object is marked with two timestamps (or version numbers) denoting the version lifespan of the textual object. Versions are retrieved in SCCS by scanning through the file and retrieving valid segments based on their timestamps. Both RCS and SCCS lack sophistication in their secondary storage management

*This research was supported by NSF grants IIS-9907477, EIA-9983445, IIS-0070135, and the Dept. of Defense.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

since they were not proposed for database applications. Moreover, neither approach supports complex queries, or queries on the evolution of the structure of the document. In fact, most of today's software configuration tools still treat a document as a sequence of lines of text, thus ignoring the rich structure of the document.

Another requirement for web-based semistructured documents is the ability of the versioning scheme to support the transport of multiversion documents across applications and to remote sites. The riches of the XML environment play a pivotal role here. 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. However, neither the edit-based, nor the time-stamped versioning schemes are conducive to our objective as they are too complex and therefore less suitable for the transport level.

To address the above requirements, this paper introduces a *reference-based* versioning scheme. The properties of the new scheme are:

1. It preserves the logical document structure across versions thus enabling efficient version retrieval and content-based querying. In addition to version reconstruction and historical queries, the scheme supports structural projection queries.
2. It is effective at the transport level. The reference-based scheme allows the whole history of a multiversioned document to be represented as yet another regular XML document.
3. It is efficient at the storage level. This is achieved by extending our reference-based scheme with storage techniques based on the notion of *page usefulness*; similar techniques (e.g. "single version current utilization") are used by transaction-time databases [1, 12, 18, 19] for clustering temporal information.

The rest of the paper is organized as follows. The reference-based versioning scheme is introduced in Section 2. Section 3 examines how various version related queries are efficiently evaluated upon the new scheme. Section 4 discusses the transport level support and shows the scheme's XML representation. In Section 5 we propose a technique for the efficient storage and retrieval of documents. We compare our scheme against previous techniques, like the multiversion B-tree [1] and a Partially Persistent List (Section 6), and show its effectiveness in Section 7. Conclusions are given in Section 8.

2 The Reference-Based Scheme

In our previous work [6], we proposed techniques for the efficient storage and retrieval of multiversion documents represented by edit-based schemes. However, we soon realized the limitations of this popular version scheme. In fact, except retrieving version difference, it does not support queries well in general, since the information is split between the actual database and the script. Furthermore, the edit script represents a special object that cannot be easily accomodated at the transport level without XML extensions.

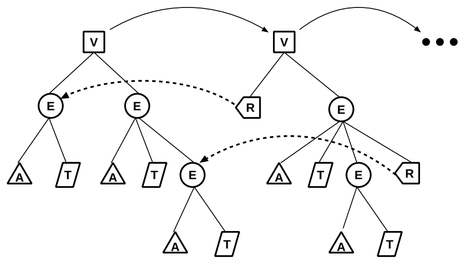

Figure 1: An $RBVM$ tree; T stands for a text node, V a version node, E an element node, R a reference node, and A an attribute node.

The reference-based representation to be discussed next, solves these logical problems without performance drawbacks.

2.1 The Reference-Based Version Model ($RBVM$)

Background. In the following discussion, we describe changes between versions by five commonly used tree-edit operations, namely, INSERT a subtree, DELETE a subtree, UPDATE the content of a node, COPY a subtree, and MOVE a subtree [23].We also define that an element is *changed* if its textual content is updated, or any of its sub-elements is *changed*. Otherwise, the element is *unchanged*. A *maximum unchanged element* is one which itself is unchanged but its parent node is changed.

The Model. The basic ideas behind $RBVM$ are :

- *keep a view for each version*
- *share unchanged elements among versions*

That is, when representing a new version, each *maximum unchanged element* is represented by a reference record, and that reference record refers to the logical location of that unchanged element in the previous version.

Thus $RBVM$ represents a simple extension with respect to the data model described in the XPath specification [20], in which XML documents are modeled as ordered-trees containing various types of nodes. To simplify the following discussion (without loss of generality), we consider only the following nodes: *element nodes*, *text nodes*, and *attribute nodes*. In addition, we introduce *version nodes* and *reference nodes*, which, respectively, serve as root nodes of versions and references to maximum unchanged elements.

The $RBVM$ models each version of an XML document as an ordered tree which contains: *element*, *text*, *attribute* and $reference$ nodes, and is rooted with a version node. A series of versions are modeled as an ordered forest of version trees as illustrated in Figure 1. Version sharing has also been proposed in [4, 3] where

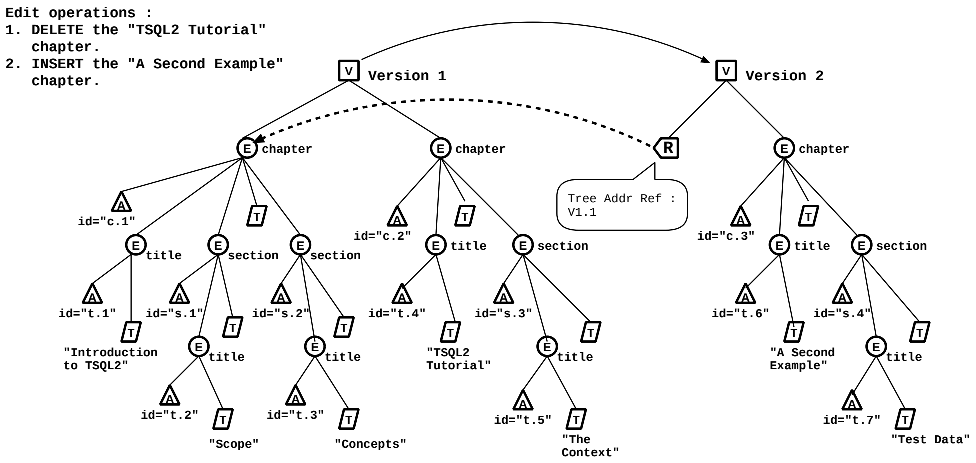

Figure 2: Sample $RBVM$ versions.

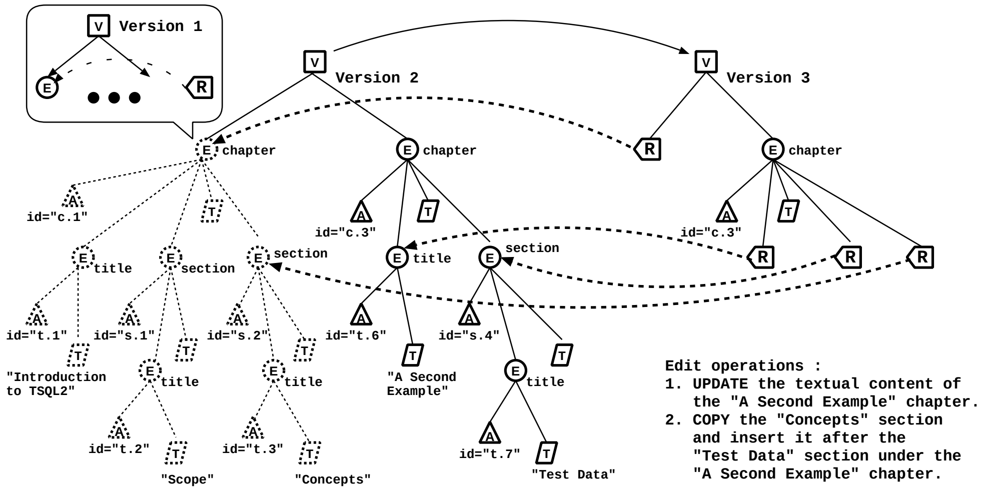

Figure 3: Sample $RBVM$ versions (cont'd).

a B+-tree is used to store the pages of a large object and subsequent versions share common paths of this B+-tree. The RBVM is at the logical level and 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.

Example. An example is given in Figures 2 and 3. Figure 2 shows the first two versions of a sample XML document. The initial version, Version 1, contains two chapters — "Introduction to TSQL2", and "TQL2 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 the "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 contains the tree address of the unchanged chapter in Version 1 — $V1.1$ — which refers to the first element at the first level of Version 1. 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 the deleted chapter does not need any handling because it does not exist in Version 2.

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 Figure 3, the "Introduction to TSQL2" chapter remains unchanged, thus, is represented by a reference record — $(V2.1)$. Note that *reference records*

```
PROCESS(K,J) {
  if (K<M) then current[K] = J;
  for (each element I from current[K] upto J) {
    if (islocal(I))
      actual_obj = fetch(I);
    else
      actual_obj = ASK(K-1, ref(I));
    if (K>=M) then output actual_obj to Version[K]}
  current[K] = J;
  return actual_obj };

ASK(K1,I) {
  return PROCESS(K1, I)};
```

Figure 4: Recursive Procedures for Materialization.

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, it contains three reference nodes and each of them refers 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. Last, a link is built from Version 2 to Version 3.

Restructuring and Duplicating. It is often the case that two sections of the old version are switched in a new version. Also some passages and footnotes might be repeated at various points in the 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.

2.2 Version Materialization

Materializing a sequence of versions, from Version M to Version N where $M \leq N$, starts from Version N. The elements of Version N are scanned sequentially starting from its root. Each version element that is stored locally is directly output. For an element that is a reference node, the previous version, i.e., Version $N-1$, is asked for the actual element. If Version $N-1$ has stored the element locally, it returns the element to Version N. Otherwise the previous version is recursively asked until the actual element is found. If the actual element is found in Version I, $M \leq I < N$, it will also belong to all versions between I and N. As a side effect, when Version N is fully materialized, involved intermediate versions have also been partially materialized.

The overall materialization of versions between M and N can be expressed as follows:

```
for (K=N; K>=M; K= K-1)
          PROCESS(K, last(K));
```

Procedure PROCESS is shown in Figure 4; $last(K)$ denotes the tree address of the last element in Version K.

In Figure 4, K and M are version numbers, while I, J, and $current[K]$ hold tree addresses for a given version. $current[]$ is a global array and it holds a tree address for each version which serves as a pointer of processing progress. Thus, for each version K, $M \leq K \leq N$, $Version[K]$ denotes the current version being assembled. Tree addresses for a given version are totally ordered. Thus, a call to $PROCESS(K, J)$ for $K \geq M$ results in materializing and storing in $Version(K)$ each element from $current[K]$ up to J; the value of $current[K]$ is also reset to J (to be used by later calls to $PROCESS(K, \_)$ that resumes the materialization of Version K).

To obtain the object with tree address I, $PROCESS(K, J)$ checks if I is actually stored in the current working version, Version K. If so, it stores it in $actual\_obj$. Otherwise, I holds a tree reference to a previous version object—which we denote by $ref(I)$. Then, the previous version is asked for this object, by calling $ASK(K-1, ref(I))$, and storing the result in $actual\_obj$. Observe that $ASK(K-1, ref(I))$ materializes Version $K-1$ from $current[K-1]$ (left by the previous call of $PROCESS$ on this version) to $ref(I)$.

If a request for element I is served by a version earlier than M, the materialization continues before version M (version $M-1$, etc.) but only elements requested by versions M through N are materialized. The first line in the routine $PROCESS$ deals with this situation. As version $M-1$ is not fully materialized, the page holding the tree address of element I (or its reference record) is needed. Since elements in a version are stored by document order, the pages of each version can be indexed by a sparse append-only index. This index maintains the tree address of the first element in each page and can be used to locate the tree address of element I.

3 Queries on Versioned Documents

There has been much interest in query languages for XML and semistructured documents. Our focus here is to evaluate the effectiveness of our $RBVM$ model to support the different kinds of queries that arise naturally for versioned XML documents instead of designing a new query language. 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 (temporal snapshot) in its entirety, or successive versions of the same—e.g., retrieve version 9 to 17. An algorithm for the efficient support of this query was presented in the previous section.
- *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 of this query kind.

• *Structural Projection.* For instance, if the user requests Section 4.1, Subsection 4.2.2, and Chapter 6 from a document, we can represent this request by the following *projection list*: PL = [4.1, 4.2.2, 6]. In a projection list, we assume that the tree addresses are listed in their natural order, and they *do not overlap.* Thus, e.g., if the list contains node 4.1, neither 4.1, nor its descendants (such as, 4.1.3, 4.1.4.2) can appear later in the projection list.

If we view successive versions as successive textual lines on a screen (of unbounded width), then the structured projection can be viewed as elimination of vertical columns, as in projections for the relational data model. (But this analogy holds only to a point, since, say, Section 4.1 in one version might be shifted to the right (left) in the next version by the insertion (deletion) of, say, Section 3.9.) 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. An efficient algorithm for Structural Projection is given below.

• *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 PL is given consisting of a sequence of non-overlapping tree addresses. We want to materialize a structured projection according to this list for versions M to N. We will keep a request list $RL(K)$. Then, the initialization step sets $RL(K) = PL$ for $M \leq K \leq N$; for every version before M, i.e., for $K < M$, $RL(K)$ is initialized to the empty list. During processing, we add to $RL(K)$ the tree addresses for the elements requests by the next version. The addition of a tree address I to a sorted projection list L leaves L unchanged when L contains I, or one of tree ancestors (super-objects); otherwise I is inserted into this sorted list. If L contains proper sub-objects (tree descendants) of I these are removed from L.

For instance, the addition of 4.1.3 to [4.1, 4.2.2, 6] leaves this list unchanged. But the addition of 4.2 to the list changes it to [4.1, 4.2, 6].

Observe the similarity between the recursive procedures in Figure 4 and those of Figure 5. However, in the former, we visit directly the elements in the version, while here we sequentially process the tree addresses in $RL(K)$ (by the statement `for each address I in RL(K) up to J`). Therefore we access the element version with address I (using the sparse index); three cases are now possible.

1. Version K holds element I. In this case $islocal(I)$ is true and $fetch(I)$ returns the actual object.

2. Version K represents element I or a super-object of

```
SP_PROCESS(K,J) {
  for (each address I in RL(K) up to J) {
    if (ismix(I)) then
       replace I in RL[K] by its children;
       I = firstchild(I) }
    else {
      if (islocal(I)) then actual_obj = fetch(I);
      if (isref(I)) then actual_obj =
        SP_ASK(K-1, refval(I)); }
    delete I from RL[K];
    if (K >= M) then
      Output sub-objects of actual_obj
      which are in PL[K];
  }
  return actual_obj;
}
SP_ASK(K1,I) {
  add I to RL[K1];
  return SP_PROCESS(K1, I) }
```

Figure 5: Recursive Procedures for Projection.

I via a reference to the previous version. In this case, $isref(I)$ is true. For instance, say that $I = 4.5.3$ and version K represents object 4 as a reference to object 6.2 in version $K-1$. Then, the element 4.5.3 of K is actually element 6.2.5.3 in version $K-1$. The function $refval$ is used to implement this mapping; in this case $refval(4.5.3) = 6.2.5.3$.

3. When neither of the two previous cases occur, then object K is a mixture of some locally stored sub-objects and references—in this case, $ismix(I)$ is true. Then, we must decompose element I into its sub-elements (children) and process them individually. We do that by replacing I in $RK(L)$ by its children and resuming from its first child, denoted $firstchild(I)$.

For versions between M and N once an object for the request list is materialized we output every sub-object of *actual_obj* which was in the original list.

The overall materialization of projections M and N requires the initialization of the $RL(K)$ lists followed by the invocation of $SP\_PROCESS(K, last(K))$ starting from N and going back to M

Content-Based Selection. In many queries this is also combined with structural projections. Selection is easily performed on versions (or their structured projections) materialized by the algorithms previously discussed. Further optimization is also possible by pushing selection onto the referenced elements of the previous version; this is an interesting problem but due to space limitations details are omitted.

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 the 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 $RBVM$ model, the first phase is no longer necessary, since that information is already

```
     <!-- 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 RefRecord>
     <!ATTLIST RefRecord v_number CDATA>
     <!ATTLIST RefRecord start_element IDREF>
     <!ATTLIST RefRecord end_element IDREF>
N.4  <!ELEMENT OrdinaryIssuePage
       (volume_info,sectionList)>
N.5  <!ELEMENT volume_info ((#PCDATA)|RefRecord)>
     <!ATTLIST volumn_info id ID>
N.6  <!ELEMENT sectionList
       ((sectionListTuple)* | RefRecord)>
     <!ATTLIST sectionList id ID>
N.7  <!ELEMENT sectionListTuple
       ((sectionName,articles) | RefRecord)>
     <!ATTLIST sectionListTuplle id ID>
     ...
```

Figure 6: A sample DTD and its version DTD.

embedded in the references (each reference denotes a matching segment). Therefore, generating version difference for $RBVM$ only requires a one-pass bottom-up breadth-first search over Version M and its previous version.

In summary, $RBVM$ provides an excellent basis for supporting the assortment of queries of different types that are typical on versioned XML documents.

4 Transport Level: RBVM in XML

Since the reference-based model preserves the logical structure of each version, it is perfectly suited to support transport level. With a version DTD, the content of a reference-based version repository can be represented in XML and seamlessly viewed, retrieved, transported, or restructured by applications which understand the DTD. In the following, we discuss how to derive a DTD for the version repository based on its original document DTD.

Since $RBVM$ itself has 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 steps :

- 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 6 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 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, $OrdinaryIssuePage$, is unchanged while, for the other elements, their content model is extended to include the $RefRecord$ element as an alternate. Indeed, the occurrences of such elements are reference records instead of the actual objects when they remained unchanged from the previous version. For example, the *sectionList* element defined at G.3 of the original DTD is changed to include a $RefRecord$ element as an alternate content. The result version DTD is shown in Figure 6. Based on the above procedure, the version DTD can be automatically derived from the original DTD and be used by any application which understand the content of the repository.

Recently, a schema definition language, XML Schema [22], was proposed by the World Wide Web Consortium (W3C) to support richer semantics for XML documents. The previous procedure to derive a version DTD from the original DTD can be easily extended to derive a Version schema for any XML document defined by a schema. The procedure is as follows:

- Define new element types for repository, versions, and reference records.
- For each element type defined in the original schema, except the root element, change its content model to a *union type* whose member types consist of the original content model and the *reference record type*. That indicates the fact that the occurrence of the element may be a reference record if it is unchanged from the previous version.
- Define an ID attribute to each element type that does not have one.

5 Usefulness-Based Storage Scheme

In order to improve performance, we develop algorithms that enhance the above reference-based scheme with a *usefulness-based* clustering scheme. The usefulness-based clustering is an extension of the clustering schemes found in temporal databases (e.g. version utilization [12] or page usefulness [18]).

At the storage level, the reference-based scheme stores both the reference and actual object nodes of a version in data pages sequentially by their document order (the order in which the first character of the XML representation of each node appears in the XML representation of the version). Along the evolution process, the number of valid objects data pages contain may decrease because some objects are deleted.

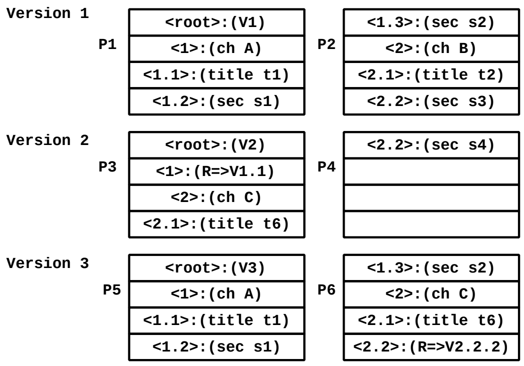

Figure 7: A sample $RBVM$ version repository.

This process will make a later version have its valid objects sparsely scattered among data pages. As a result, materializing such a version needs to access more pages and, thus, the I/O cost increases. To solve this problem, sparsely scattered objects should be clustered together. In the following we describe a mechanism which clusters objects in in a controlled way.

Figure 7 shows a simplified example from Figure 2 3 where all text nodes, and attribute nodes are ignored as well as those element nodes at the bottom level. Each object starts with its tree address. If the object is a reference record it is denoted as (R⇒T) where T is the tree address of the referred unchanged element. Otherwise, it is an actual object and stored locally. In this example, we assume that (1) reference records have the same size as the actual object records (whereas they are normally smaller), and, (2) a page holds four (object or reference) records.

Let us start with the definition of the *segmental usefulness* of a data page.

Definition: Consider page P and let S be the logical segment it contains for version V. Let pages $P_1, P_2, \cdots, P_n$ be the extra pages needed to be accessed for materializing S, and let B be the page size. The segmental usefulness of page P for version V is the ratio of the size of S over the total size of pages $P, P_1, P_2, \cdots, P_n$:

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

□

Version 1 has all actual objects, so its objects are stored by the document order in data pages P1 and P2 as shown in Figure 7. Pages P1 and P2 both have segmental usefulness of 100% because the nodes they contain are all stored locally.

The representation of Version 2 contains both reference records and actual objects. The first four records are stored in a new data page P3. Consider the overall "logical" segment corresponding to page P3, which we will denote S_3. This logical segment starts from the root object in Version 2 and extends until the 7^{th} object of Version 2 (title t6). Note that, S_3 is comprised of three acutal elements, root, ch C and title t6, and one reference record, (R⇒V1.1). The tree address V1.1 refers to the sub-tree rooted at ch A of Version 1 which consists of three elements, ch A, title t1, and sec s1 and are stored in page P1. Therefore, to physically materialize S_3, pages P1 and P2 must also be accessed. Out of the twelve records read (i.e., the records in pages P1, P2 and P3) seven correspond to real objects (i.e., "useful" objects), one is a reference record (from page P3), and three are expired (ch B and its two sub-elements). Collectively, 58% of three accessed data pages, P3, P1 and P2, are *useful* for S_3. That is, the *segmental usefulness* of page P3 is 58%. The last node of Version 2, (sec s4), is stored in the next new page, P4. Since page P4 is the end page of Version 2 and its only object is actual object, it is considered 100%. For each version, there exists only one end page.

The usefulness of a page represents the I/O efficiency of materializing the overall logical segment it contains. If the usefulness of a page is U and its logical segment is S, the total size of data pages accessed for materializing S is $size(S)/U$. The 58% segmental usefulness of page P3 implies that the total number of records accessed for materializing S_3 is 172% of the size of S_3.

To guarantee low I/O cost, we define a minimum required usefulness U_{min}. A page will be called $useful$ as long as its usefulness is greater or equal than U_{min}. The next question is what to do with pages whose usefulness is below U_{min}. For example, assume U_{min} has been set to 50% and now consider Version 3 whose reference-base representation is :

$$< root >: (V3), < 1 >: (R \Rightarrow V2.1),$$
$$< 2 >: (R \Rightarrow V2.2), < 2.1 >: (R \Rightarrow V2.2.1),$$
$$< 2.2 >: (R \Rightarrow V2.2.2), < 2.3 >: (R \Rightarrow V2.1.3).$$

If the first four nodes were stored in a new page, say page P', materializing this segment in P' requires reading pages P', P3, P1, and P2. However, only seven out of the sixteen objects read are useful; that is, P' has segmental usefulness only 43.75% which is below U_{min}. The cost of materializing the segment of P' is high: 229%, To avoid high I/O cost, it is better to materialize the actual objects of a useless page and cluster them together in pages. This effectively creates copies of actual records.

Copy Procedure. Instead of storing page P', the version materialization algorithms in Figure 4 — PROCESS() and ASK() — are used to identify the actual objects of the logical segment in P'. These objects are copied and stored in new data pages P5 and P6 as shown in Figure 7. Since page P6 is not full, the next reference record, $< 2.2 >: (R \Rightarrow V2.2.2)$, is stored after the last copied object, (title t6). The usefulness of page P6 is 50% because materializing its segment involves one extra page P4 and four objects out of these two pages are useful. So, P6 is useful and kept intact.

Complexity Analysis. In [7] it is shown that the space used by our scheme is bound by $(O(S_{chg}))$, where

S_{chg} is the total number of version changes The cost for reconstructing any given version can be computed from the number of pages that have to be read into main memory. Reconstruction of a version of size S_i needs $\frac{1}{U_{min}} \times \frac{S_i}{B}$ pages, where B is the capacity of the page [7]. Last, the complete version insertion algorithm is the following :

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

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 might stop the materialization as soon as the first page boundary is reached, and return to step 2 at that point; the second page and the other pages might in fact be still useful.

6 Timestamp-Based Schemes

A document can be visualized as an ordered list of its element objects. This logical order need be preserved when reconstructing a version of the document. Hence, when physically storing a document we can utilize a data structure that maintains such an order (for example a B+-tree or an ordered list). Document versioning is then reduced to making this data structure partially persistent. A data structure is called persistent if an update creates a new version of the data structure while the previous version is still retained and can be accessed (if the old version is discarded, the structure is called ephemeral). Partial persistence implies that updates are applied only at the latest version of the data structure.

6.1 Utilizing a Multiversion B-Tree

Assume that we have a B+-tree that indexes the object positions in the first version of a document. That is, the leaves of this B+-tree contain records with keys 1, 2, 3, etc. where record k stores the object in the $k-th$ position of the document. However, since each object insertion/deletion affects the position number of all the objects after it, updating this B+-tree becomes very inefficient. Adding/deleting one object in the beginning of the document would update all the positions after this object. This problem can be resolved if the object positions are encoded in a way not altered by document changes. One simplistic and straightforward solution is to identify object positions by an ordered sequence of large non-consecutive integers. Then a future insertion between positions x and y can be indexed by a number that lies between x and y. The choice of numbers as well as the scheme to associate new numbers for future insertions depends on the document evolution.

There have been various approaches to making a B+-tree partially persistent [1] [12] [19]. In our experiments we used the MVBT since its code is publicly available. The MVBT also uses a notion of usefulness. 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). The space used by the MVBT is linear to the number of changes S_{chg}. If version V_i had S_i objects, then the MVBT reconstructs it with $O(log_B(S_{chg}/B) + S_i/B)$ I/O's [1].

6.2 Utilizing a Partially Persistent List

Let L be an ephemeral list that maintains the relative positions of its elements. 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. A page is useful for a given version as long as it has enough valid records from this version. 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. Since list reconstruction starts from the top element in the list, the first page of L must be identified for any given version. Furthermore, a useful page should also maintain the next useful page in the list (which may change many times for a given page). In [7] we present a partially persistent list that uses linear space and reconstructs version V_i with $O(log_B(S_{chg}/B) + S_i/B)$ I/O's.

7 Performance Analysis

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

We first compare the behavior of all schemes under the same usefulness requirement. For this experiment we used a document evolution with the following characteristics : 1) the size of each version is approximately 100 pages; 2) each version changes about 20% from the previous version (half of the changes are insertions and the other half are deletions); 3) the usefulness parameter is set to 50%; 4) the document evolution had a total

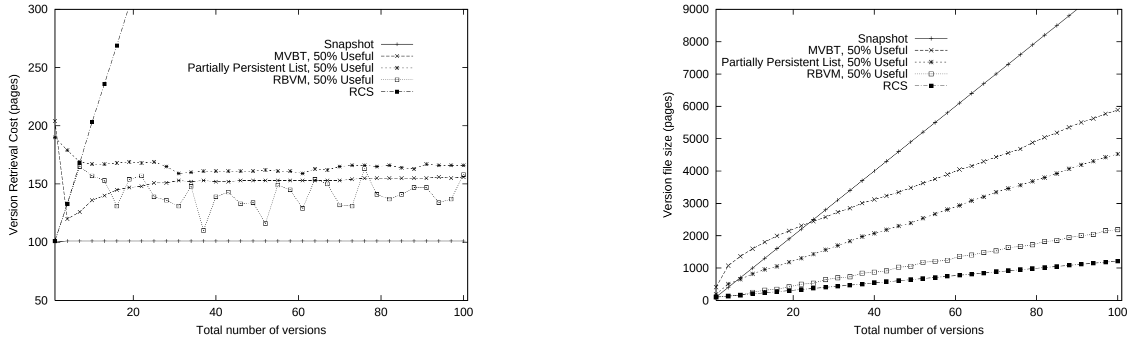

Figure 8: Version retrieval and storage cost with 50% usefulness requirement.

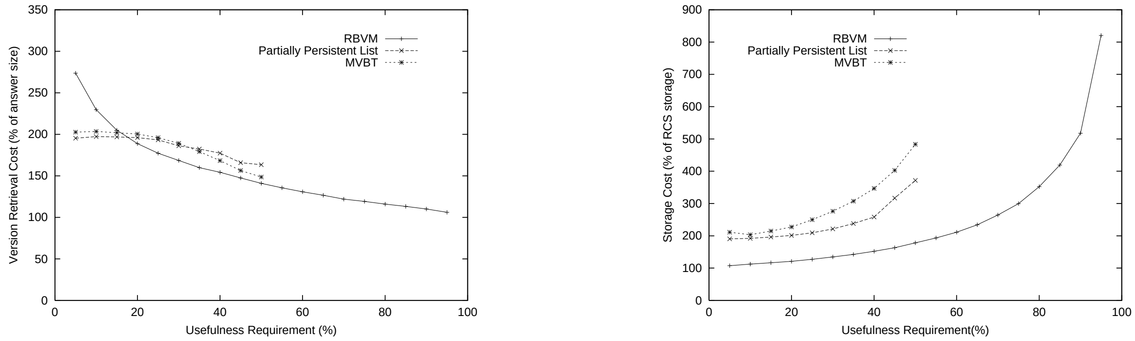

Figure 9: Version retrieval and storage cost v.s. Usefulness

of 100 versions. Changes are uniformly and randomly distributed among data pages.

Figure 8 shows the version retrieval cost measured as the number of page I/O's needed to reconstruct a version. The "Snapshot" scheme clearly has the minimal version retrieval cost, since each version is already stored in its entirety on disk. As expected, all page-usefulness schemes have version retrieval cost that is proportional to the size of the reconstructed version. (In this experiment, the average version size remains the same –about 100 pages–, so the retrieval cost of the three page-usefulness schemes and the Snapshot scheme are approximately parallel to the horizontal axis). The retrieval overhead against the Snapshot scheme is because a useful page includes some non-valid objects. Thus RBVM, MVBT and PPL have to access more pages than the Snapshot scheme. However, this overhead is constant. In particular, the RBVM, in average, has the best retrieval performance among the three page-usefulness schemes. In addition, in the RBVM scheme when the usefulness of a segment falls below the threshold, several new pages are normally generated. Because of this larger granularity, the retrieval cost shows more substantial fluctuations below MVBT and PPL. The MVBT is slightly better than the PPL because PPL uses some page capacity to identify the next-page in the list. The RCS strategy needs to read the whole database prior to the target version. Therefore, retrieving later versions gets more expensive than earlier versions.

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

The effect of usefulness. To study the effect of the usefulness parameter we run the above document evolution using different U_{min}. The experimental results are illustrated in Figure 9. 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 RBVM scheme can achieve higher usefulness. The version retrieval cost is depicted as the percentage of the answer size. 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's. Note that the RBVM scheme fills up a new page with records without reserving space for future insertions in this page. As a result, the usefulness of a RBVM page can only decrease due to record deletions. A small U implies that a page will be considered useful even if it has very few valid records. For RBVM this means that many pages may have low usefulness because of deletions. Since these pages are still useful, they are not copied. Hence, the answer will be clustered in many RBVM 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 9 depicts the storage cost as a percentage over the (minimal) RCS storage.

Setting the usefulness parameter serves as an optimization tool for each of the three schemes in the presence of limited storage. For instance, if only 200% extra space is available, the MVBT scheme can guarantee 35% usefulness, PPL 45% usefulness, while the RBVM 75% usefulness. Choosing higher usefulness (RBVM) is definitely preferable since the retrieval time will be better.

Increasing and Decreasing Document Sizes. We further examined evolutions where the document size consistently increases or decreases. The results of these experiments also show that the RBVM scheme performs better than both the MVBT scheme and the PPL scheme. Experiment results are available at [7].

The page-usefulness schemes provide a more robust behavior than traditional approaches (e.g. RCS). At the expense of some (linear) extra space, the version retrieval cost is proportional to the target version size. Among them, the RBVM scheme provides the superior performance that can be easily tuned through the usefulness parameter (depending on the characteristics of the document evolution or users' access pattern).

8 Conclusions

In this paper we introduced a versioning scheme for semi-structured document and showed its effectiveness at both the transport and storage levels. Moreover, the presented scheme supports retrieval of complete versions and content-based queries.

References

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, *"An Asymptotically Optimal Multiversion B-Tree"*, VLDB Journal, Vol.5, No.4, 1996, pp.264-275.

[2] D. Beech, B. Mahbod, *"Generalized Version Control in an Object-Oriented Database"*, IEEE Fourth International Conference on Data Engineering, Feb. 1988.

[3] F. Warren Burton, John G. Kollias, D. G. Matsakis, V. G. Kollias, *Implementation of Overlapping B-Trees for Time and Space Efficient Representation of Collections of Similar Files*, The Computer Journal 33(3): 279-280 (1990).

[4] M.J. Carey, D.J. DeWitt, J.E. Richardson and E.J. Shekita, *Object and File Management in the EXODUS Extensible Database System*, In Proc. VLDB Conference, pp. 91-100, 1986.

[5] S. Chawathe, H. Garcia-Molina, *"Representing and Querying changes in semistructured data"*, Proc. of the International Conference on Data Engineering, 1998, pp. 4-13.

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

[7] S.-Y. Chien, V.J. Tsotras, and C. Zaniolo, *Efficient Management of Multiversion Documents by Object Referencing*, UCLA Tech.Rep.No 010024, June 2001.

[8] H.-T. Chou, W. Kim, *A Unifying Framework for Version Control in a CAD Environment*, In Proc. of VLDB Conf., Kyoto, Japan, 1986.

[9] M. C. Easton, *Key-Sequence Data Sets on Inedible Storage*, IBM Journal of Research and Development 30(3): 230-241 (1986).

[10] R.H. Katz, E. Change, *"Managing Change in Computer-Aided Design Databases"*, Proc. of VLDB Conf., Brighton, England, Sep. 1987.

[11] D. B. Leblang *The CM Challenge: Configuration Management that Works*, Configuration Management, ed. by Walter F. Tichy. Published by Wiley Co. 1994, pp.1-38.

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

[13] A. Marian, S. Abiteboul, G. Cobena and L. Mignet, *Change-centric management of versions in an XML warehouse* In Proc. of the 27th VLDB, Rome, Italy, Sep. 2001.

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

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

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

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

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

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

[20] World Wide Web Consortium, *XML Path Language (XPath)* Version 1.0. Nov. 16, 1999. See http: //www. w3. org/ TR/ xpath.html

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

[22] XML Schema, World Wide Web Consortium. See http://www. w3. org/ XML/ Schema.

[23] K. Zhang, *Algorithms For The Constrained Editing Distance Between Ordered Labeled Trees And Related Problems*, Pattern Recognition, vol.28, no.3, pp.463-474, 1995.