Storing and Querying Multiversion XML Documents using Durable Node Numbers

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

Vassilis J. Tsotras
Dept. of CS&E
UC Riverside
tsotras@cs.ucr.edu

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

Donghui Zhang
Dept. of CS&E
UC Riverside
donghui@cs.ucr.edu

Abstract

Managing multiple versions of XML documents represents an important problem for many traditional applications, such as software configuration control, as well as new ones, such as link permanence of web documents. Research on managing multiversion XML documents seeks to provide efficient and robust techniques for storing, retrieving and querying such documents. In this paper, we present a novel approach to version management that achieves these objectives by a scheme based on Durable Node Numbers and timestamps for the elements of XML documents. We first present efficient storage and retrieval techniques for multiversion documents. Then, we explore the indexing and clustering strategies needed to assure efficient support for complex queries on content and on document evolution.

1 Introduction

The management of multiple versions of XML documents finds important applications [21] and poses interesting technical challenges. Indeed, the problem is important for application domains, such as software configuration and cooperative work, that have traditionally relied on version management. As these applications migrate to a web-based environment, they are increasingly using XML for representing and exchanging information—often seeking standard vendor-supported tools and environments for processing and exchanging their XML documents.

Many new applications of versioning are also emerging because of the web; a particularly important and pervasive one is assuring link permanence for web documents. Any URL becoming invalid causes serious problems for all documents referring to it—a problem that is particularly severe for search engines that risk directing 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. For this reason, professionally managed sites and content providers will have to use document versioning systems; frequently, web service providers will also support searches and queries on their repositories of multiversion documents. Specialty warehouses and archives that monitor and collect content from websites of interest will also rely on versioning to preserve information, track the history of downloaded documents, and support queries on these documents and their history [11].

Current document management systems used in software configuration support and other applications rely on two versioning schemes, RCS [18] and SCCS [16], discussed next. The RCS scheme is *edit-based*: edit scripts are used to represent document changes and to reconstruct different versions incrementally. Thus RCS [18] stores the most current version intact while previous versions are stored as reverse edit scripts. These scripts describe how to go backward in the document's development history. For any version except the current one, extra processing is needed to apply the reverse editing script to generate the old version. Rather than appending version differences at the end, SCCS [16] inserts editing operations in the original (source code) document and associates a pair of timestamps (version ids) with each document segment to specify its lifespan. Versions are retrieved from an SCCS file via scanning through the file and retrieving valid segments based on their timestamps.

Various techniques for versioning have also been proposed by database researchers who have focused on problems such as transaction-time management of temporal databases [13], support for versions of CAD artifacts in object-oriented databases [15] and, more recently, change management for semistructured information [3].

In the past, the approaches to versioning taken by database systems and document management systems have often been different, because of the different requirements facing the two application areas. In fact:

Proceedings of the Second International Conference on Web Information Systems Engineering (WISE'02)
0-7695-1393-X/02 $17.00 © 2002 IEEE

- Database systems are designed to support complex queries, while document management systems are not, and
- Databases assume that the order of the objects is not significant—but the lexicographical order of the objects in a document is essential to its reconstruction.

This state of affairs has been changed dramatically by XML that merges applications, requirements and enabling technology from the two areas. Indeed the differences mentioned above are fast disappearing since:

- support for complex queries on XML documents is critical. This is demonstrated by the amount of current research on this topic [17, 22] and the emergence of powerful XML query languages [2], and
- the structure and position of their objects (i.e., its elements) is essential for XML documents and must be preserved for browsing and various processing tasks performed on the web. Furthermore, queries on XML documents might make use of this information.

In [4, 7], we extended the edit-based representation of RCS with an efficient clustering scheme (the *usefulness-based page management*) that clusters together data valid for the same version. This approach is very effective for retrieval of complete versions but does not support well more complex queries on XML documents. Preorder traversal numbers are used to identify the elements of the XML document viewed as an ordered tree. While easy to compute, these numbers do not provide for *durable referencing* by external indices since insertions and deletions in the document change the preorder numbers of all the elements that follow. However, numbering schemes have been recently proposed for XML documents, that are durable with respect to document changes [14, 10].

In this paper we propose a new document versioning scheme (SPaR) that, while preserving the page management approach of [4, 7], it does away with the edit-based representation. Our scheme uses timestamping and durable node numbers to preserve the structure and the history of the document during its evolution. Furthermore, the new scheme is conducive to efficient support of both version and content queries, using multiversion indexing techniques. The rest of the paper is organized as follows. The next section summarizes previous work while Section 3 introduces the SPaR versioning scheme. Full version reconstruction under the new scheme is described in Section 4 and more complex queries are discussed in Section 5. Preliminary performance results are presented in Section 6 while conclusions and further open problems appear in Section 7.

2 Background

A new document version (V_{j+1}) is established by applying a number of changes (object insertions, deletions or updates) to the current version (V_j). In a typical RCS scheme, these changes are stored in a (forward or reverse) edit script. Such a script could be generated directly from the edit commands of a structured editor, if one was used to revise the XML document. In most situations, however, the script will be obtained by applying, to the pair (V_j, V_{j+1}), a structured diff package [9].

For forward editing scripts, the RCS scheme stores the script and the data together is successive pages. Thus, to reconstruct version (V_j) all pages stored by successive versions up to version (V_j) must be retrieved. The SCSS tries to improve the situation by keeping an index that identifies the pages used by each version. However, as the document evolves, document objects valid for a given version can be dispersed in various disk pages. Since a given page may contain very few of the document objects for the requested version, many more pages must be accessed to reconstruct a version.

To solve these problems, in [4] we introduced an edit-based versioning scheme that (i) separates the actual document data from the edit script, and (ii) uses a usefulness-based clustering scheme for page management. Because of (i) the script is rather small and can be easily accessed. The usefulness-based clustering is similar to a technique used in transaction-time databases [12, 19, 1] for clustering temporal data.

Page Usefulness. Consider the actual document objects and their organization in disk pages. For simplicity, assume the only changes between document versions are object additions and deletions (other document changes are discussed later). As objects are added in the document, they are stored sequentially in pages. Object deletions are not physical but logical; the objects remain in the pages where they were recorded, but the script is updated marking such objects as deleted. 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 reconstruction algorithm will have to access many pages. Ideally we would like to cluster the objects valid at a given version in few, *useful* pages. 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. Hence page P is 100% useful for version V_1. Its usefulness falls to 90%

for version V_2, since object O_2 is deleted at V_2. Similarly, P is 70% useful for version V_3. For version V_4, P is only 55% useful.

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

We note that the above page usefulness definition holds for full pages. A page is called an *acceptor* for as long as document objects are stored in this page. While being the acceptor (and thus not yet full), a page is by definition useful. This is needed since an acceptor page may not be full but can still contain elements alive for the current version. Note that there is always only one acceptor page. After a page becomes full (and stops being the acceptor) it remains useful only as long as it contains enough alive elements (the U_{min} parameter).

Reconstructing a given version is then reduced to retrieving only the useful pages for this version. Various schemes can be used to assure that only useful pages are retrieved when reconstructing a version. The UBCC scheme described in [4] uses the edit script to determine the useful pages for each version— along with the order in which these pages must be accessed to reconstruct the order of the document. Techniques to keep the edit script within a small percentage of the actual data are presented in [5].

While the UBCC schema is effective at the storage level, it is not effective with complex queries and the transport level (i.e., the web-based exchange of documents). A *reference-based* scheme was thus introduced in [7] which facilitates the exchange of multiversion documents by representing the whole history of a document as yet another XML document. The reference-based scheme is also conducive to efficient support for some queries but not all; in particular, it does not support the path-expression queries discussed next.

A path-expression query is described by a regular expression on the document tree. For example, the query 'Find all figures in chapter 10 of the document' is supported in XML query languages [2] by a special path-expression notation: `chapter[10]/*/figure`. Figures may be anywhere in the subtree rooted in the `chapter[10]` node of the document. To answer such queries efficiently (i.e., without fully traversing document subtrees), a scheme is needed to quickly identify ancestor-descendant relationships between document elements. A numbering scheme is proposed in [14], whereby the number assigned to an element remains the same even though other elements are added or deleted from the document. This is achieved by sorting the nodes as in the preorder traversal, but leaving space between them to make room for future insertions.

Such a durable numbering scheme is advantageous since it automatically maintains the document tree structure. One has to maintain an ordered list with the node durable numbers. Moreover it allows indexing various document elements in a persistent way. An interesting open problem is whether an efficient versioning scheme exists for documents that use durable numbers. In the next section we propose such a scheme and discuss how complex queries involving path expressions over versions can be addressed.

3 The SPaR Versioning Scheme

The new indexing scheme relies on assigning durable structure-encoding ID numbers to the elements of the document [14]. An XML document is viewed as an ordered tree, where the tree nodes corresponds to document elements (and the two terms will be used as synonyms). A preorder traversal number can then identify uniquely the elements of the XML tree. While this is easy to compute, it does not provide a *durable reference* for external indexes and other objects that need to point to the document element, since insertions and deletions normally change the preorder numbers of the document elements which follow. Instead, we need durable node IDs that can be used as stable references in indexing the elements and will also allow the decomposition of the documents in several linked files [14]. Furthermore, these durable IDs must also describe the position of the element in the original document— a requirement not typically found for IDs in object-oriented databases.

3.1 The Numbering Scheme

The SPaR (Sparse Preorder and Range) numbering scheme consists of (1) a $Durable\ Node\ Number(DNN)$ and (2) a $Range$. The DNN establishes the same total order on the elements of the document as the preorder traversal, but, rather than using consecutive integers, leaves as much an interval between nodes as possible; thus DNN is a sparse numbering scheme that preserves the lexicographical order of the document elements.

The second element in the SPaR scheme is the Range. This was proposed in [14] as a mechanism for supporting efficiently *path expression queries*. For instance, a document might have chapter elements and figure elements contained in such chapters. Thus, a typical query might be *'Retrieve all titles under chapter elements'*. Recently proposed XML query languages [2] provide a special path expression to support these queries, as follows:

`doc/chapter/*/figure`

Let $dnn(E)$ and $range(E)$ denote the DNN and the range of a given element E; then a node B is descendant of a node A [1] iff:

$$dnn(A) \leq dnn(B) \leq dnn(A) + range(A).$$

Therefore, the interval $[dnn(X), dnn(X) + range(X)]$ is associated with element X. When the elements in the document are updated, their SPaR numbers remain unchanged. When new elements are inserted, they are assigned a DNN and a range that do not interfere with the SPaR of their neighbors—actually, we want to maintain sparsity by keeping the intervals of nearby nodes as far apart as possible.

Consider two consecutive document elements X and Z where $dnn(X) < dnn(Z)$. Then element Z can either be (i) the first child of X, (ii) the next sibling of X, or (iii) the next sibling of an element K who is an ancestor of X. If a new element Y is inserted between elements X and Z, it can similarly be the first child of X, the next sibling of X or the next sibling of one of X's ancestors. For each of these three cases, the location of Z creates three subcases, for a total of nine possibilities. For simplicity we discuss the insertion of Y as the first child of X and consider the possible locations for element Z (the other cases are treated similarly). Then we have that:

1. Z becomes the first child of Y. In this case the following conditions should hold: $dnn(X) < dnn(Y) < dnn(Z)$ and $dnn(Z) + range(Z) \leq dnn(Y) + range(Y) \leq dnn(X) + range(X)$.
2. Z becomes the next sibling of Y under X. The interval of new element Y is inserted in the middle of the empty interval between $dnn(X)$ and $dnn(Z)$ (thus, the conditions $dnn(X) < dnn(Y)$ and $dnn(Y) + range(Y) \leq dnn(Z)$ must hold).
3. Z becomes the next sibling of an ancestor of Y. Then element Y is "covered" by element X which implies that: $dnn(X) < dnn(Y)$ and $dnn(Y) + range(Y) \leq dnn(X) + range(X)$.

Our insertion scheme assumes that an empty interval is at hand for every new element being inserted. When integers are used, **range overflow** may occur occasionally, thus, SPaR reassignments might be needed to assure this property. A better solution is to use *floating point numbers* with *variable length*, where additional decimal digits can be added as needed for new insertions. For instance, let elements X and Z be siblings, with: $dnn(X) + range(X) = .22$ and $dnn(Z) = .23$. If element Y is to be inserted between X and Z, we can set $dnn(Y) = .222$ and $range(Y) = .006$ since:

$$dnn(X) + range(X) = .22 < dnn(Y) = .222 < dnn(Y) + range(Y) = .228 < dnn(Z) = .23$$

Nevertheless, for simplicity of exposition, in the following examples we use integer SPaRs.

[1] If the preorder traversal number is used as DNN, then $range(A)$ is equal to the number of descendants of A.

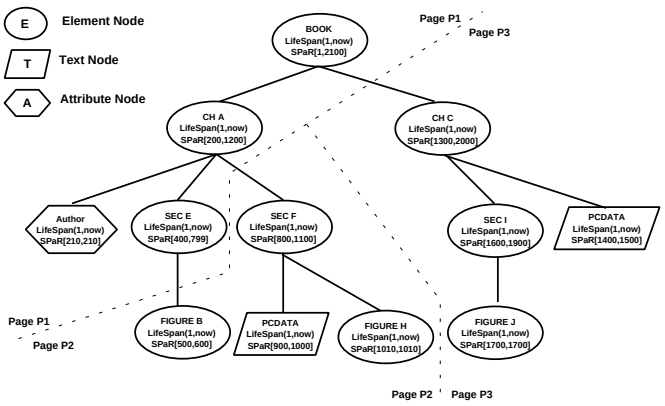

Figure 1. An XML document version represented in the SPaR model.

Figure 1 shows a sample XML document with its SPaR values. The root element is assigned range [1,2100]. That range is split into five sub-ranges — [1,199], [200,1200], [1201,1299], [1300,2000], and [2001,2100] for its two direct child elements, CH A and CH C, and three insertion points, before CH A, after CH A and before CH C and after CH C. The range assigned to each of these chapter element continues to be split and assigned to their direct child elements until leaf elements are met.

The SPaR numbering scheme makes it possible to use timestamps to manage changes in both the content and the structure of documents. In the next section, we describe a new model which uses SPaR values and timestamps to manage XML document versions.

3.2 The Version Model

The record of each XML document element is extended with the element's SPaR and the element's lifespan. The lifespan is described by two timestamps (V_{start}, V_{end})—where V_{start} is the version where the element is created and V_{end} is the version where the element is deleted. An element is called "alive" for all versions in its lifespan. If an element is inserted in the current version, its V_{end} value is yet unknown and is represented by variable now. This variable corresponds to the ever increasing current version (as per the standard notion from temporal databases [8]).

Adding a New Version The elements of the initial version are stored in new data pages by their document (DNN) order. We then assume that successive versions are described with respect to the last version by an edit script generated by the structured XML editor used to generate

P1	Unchanged			
P2	FIGURE B Life:(1,1) SPaR: [500,600]	SEC F Life:(1,1) SPaR: [800,1100]	PCDATA Life:(1,1) SPaR: [900,1000]	FIGURE H Life:(1,1) SPaR: [1010,1010]
P3	CH C Life:(1,1) SPaR: [1300,2000]	SEC I Life:(1,1) SPaR: [1600,1900]	FIGURE J Life:(1,1) SPaR: [1700,1700]	PCDATA Life:(1,1) SPaR: [1400,1500]
P4	FIGURE B Life:(2,now) SPaR: [500,600]	SEC F Life:(2,now) SPaR: [800,1100]	PCDATA (new) Life:(2,now) SPaR: [900,1000]	CH K Life:(2,now) SPaR: [1420,1480]
P5	CH C Life:(2,now) SPaR: [1300,2000]	SEC I Life:(2,now) SPaR: [1600,1900]	SEC J Life:(2,now) SPaR [1700,1700]	SEC M Life:(2,now) SPaR: [1930,1960]

Figure 2. The Repository for Version 2.

the new version, or otherwise by a package that computes the structured diff between the two documents. Then, the following operations are performed when in version V_N we delete, insert or update an element:

- $DELETE$ — Update the V_{end} timestamp of the deleted *element and all its descendants* as expired at Version V_N. Free the $SPaR$ range of deleted elements for reuse.
- $INSERT$ — Create a record for the new element and initialize its lifespan to (V_N, now). Assign an unused range to the new element based on the weighted range allocation algorithm. The record is stored in the current acceptor page.
- $UPDATE$ — Update the V_{end} timestamp of the updated element as expired at Version V_N. Create a new record and initialize its lifespan to (V_N, now). The new record should keep the same $SPaR$ values (since the position of the updated element in the document did not change).

Additional operations, such as $MOVE$ elements and $COPY$ elements can be reduced to the above.

Usefulness-Based Copying. Whenever a page falls below the U_{min} usefulness level all its alive elements are copied to a new page, in the order established by their SPaR values. All copied elements preserve their SPaR values, but are given a new lifespan, as if they were updated—in fact, copying can be treated as an update where the new SPaR values are the same as the old ones. Note that the delta elements for each new version (i.e., the newly inserted elements as well as the copied elements due to usefulness) are stored in pages by increasing DNN values.

Example. Elements of the initial version (Version 1), are stored with their $SPaR$ range and lifespan in pages P1, P2 and P3 as shown in Figure 1. We have assumed that the sizes of document objects, BOOK, CH A, attribute AUTHOR, SEC E, FIGURE B, SEC F, PCDATA of SEC F, FIGURE H, CH C, SEC I, FIGURE J, and PCDATA of CH C are 50%, 25%, 10%, 15%, 5%, 30%, 35%, 30%, 5%, 10%, 5%, and 80% of a data page size, respectively. Assume that we want to maintain a minimum page usefulness of 70%. Then pages P1, P2 and P3 are well above the threshold for Version 1.

Assume that Version 2 is created by the following changes:

(delete AUTHOR), (update PCDATA of SEC F), (delete FIGURE H), (insert CH K after CH A), (delete PCDATA of CH C), (insert SEC M)

Let the sizes of the new PCDATA of SEC F, CH K and SEC M be 20%, 45%, and 50% of a data page size, respectively. Hence, the logical order of objects in version 2 are: BOOK, CH A, SEC E, FIGURE B, SEC F, new PCDATA of SEC F, CH K, CH C, SEC I, FIGURE J, and SEC M. After applying these changes, Page P1 becomes 90% useful (AUTHOR is deleted for version 2), page P2 becomes 35% useful (since the old PCDATA for SEC F and FIGURE H are not part of Version 2) and page P3 becomes 20% useful because of the deletion of the PCDATA of CH C. Then, pages P2 and P3 are *useless* for the second version and, thus, valid objects in P2 and P3 are copied into a new data page. Copied objects include FIGURE B, SEC F, CH C, SEC I, and FIGURE J.

After determining which objects need copying, the copied objects are inserted into new pages together with new objects (new PCDATA of SEC F, CH K, and SEC M) in their logical DNN order as shown in Figure 2.

4 Full Version Reconstruction

Reconstructing a given document version consists of three tasks: (1) identifying the useful pages for the given version, (2) ordering the elements according to their SPaR number, and (3) reconstructing the ordered-tree structure of the document.

Identifying the Useful Pages. The notion of usefulness associates with each page a *usefulness interval*. This interval has also the form of (V_{start}, V_{end}), where V_{start} is the version when the page became acceptor and V_{end} is the version when the page became non-useful. As with the element records, a page usefulness interval is initiated as (V_{start}, now) and later updated at V_{end}. Identifying the data pages that were useful in V_i is then equivalent to finding which pages have intervals that contain V_i. This problem has been solved in temporal databases with an access method called the Snapshot Index [19]. This approach uses a doubly-linked list L and an array A as described next.

List L is initiated with a special record S which always remains at the top of the list and is never removed. Let lpr

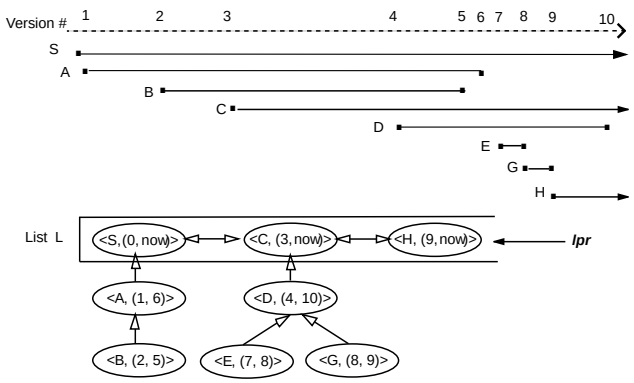

Figure 3. Snapshot Index.

be a pointer pointing to the end of L. Initially lpr points to S. When page p becomes acceptor at version V_i a record of the form $(p, (V_i, now))$ is added at the end of L and lpr points to this new record. Similarly, records are added at the end of the list for each acceptor page. When page p becomes non-useful, three actions are taken: (i) the usefulness interval of p's record is updated, (ii) the record is taken off the list and (iii) it is assigned as the next child under the record that was before p in the list.

The above procedure creates a structure that has the form of trees rooted in records that are still located in list L (the $access\ forest$). The access forest has the following properties ([19]): (1) For each version, list L contains the useful pages in this version. (2) In each tree in the access forest, the V_{start} fields are in preorder. (3) The usefulness interval of a parent record contains all the usefulness intervals of the records in its subtree. (4) The intervals (V_i, V_j) and (V_{i+1}, V_{j+1}) of two consecutive records under the same parent may have one of the following orderings: $V_i < V_j < V_{i+1} < V_{j+1}$, or, $V_i < V_{i+1} < V_j < V_{j+1}$. Figure 3 presents the usefulness intervals for various pages and the corresponding access forest created through list L. The current version is version 10. Since S was in the list before any other record, its interval is $(0, now)$. Using these properties, the pages useful at a given version can be easily identified.

The array A stores, for each version, a pointer to the record of the last acceptor page used by this version. That is, array records have the form: $(version-id, pointer)$. Array A is incrementally updated and is easily maintained (using for example a B+-tree on the version numbers).

Given a version V_i, we identify from array A the record, say x_i, in the access forest where the search starts. Since x_i corresponds to an acceptor page, it was useful for version V_i. Then all the records in the path from x_i to a root (a record that is still in L) correspond also to useful pages in version V_i. For each useful record, one needs to check whether its left sibling and its rightmost child records are useful (i.e., they identify useful pages). The search proceeds similarly and stops when non-useful records are reached. For example, to find the useful pages at version 7 we check the records of E, D, C, S and A. Note that this algorithm checks at most two non-useful records for each useful one. If there were u_i useful pages in version V_i, the algorithm identifies them in $O(u_i)$ steps through the access forest. Searching through array A needs an additional logarithmic effort (on the number of versions).

Ordering the Elements. The useful pages contain the alive element records for version V_i. To order these records (task 2) a straightforward approach is to perform a sort over all useful pages. This however may require reading various useful pages many times, especially when not all of them can fit in main memory. A better solution, described below, uses the fact that the delta records for each version, and the pages containing these records are already stored by increasing SPaR DNN numbers as shown in Figure 2.

Important for each page is to retain the version(s) that wrote (added) elements to it. For simplicity assume that each page is written by a single version (the page's $creator$ version). In this case, the creator version of page p is the V_{start} of the p's usefulness interval. Since a given version may write many pages, p also retains the position it had among all pages written by p's creator version. Let $V_c(i)$ be the set of creator versions for the useful pages in version V_i. For each version in $V_c(i)$ we build an ordered list with the subset of the u_i pages this version wrote. To order the alive elements in version V_i, we simply retrieve the first page from each list and start ordering the elements in a sort-merge approach. Assuming that there is enough memory to hold one page per list, this scheme sorts all alive elements in V_i by reading each useful page only once. Otherwise, standard external sorting techniques can be used.

Reconstructing the Document Structure. To reconstruct the ordered-tree structure from an ordered list of elements (task 3) we need to determine two relationships among elements: 1) parent-child relation, and 2) sibling order. This can be easily done by using the SPaR ranges and a **backward ancestor stack**. We use the stack to record the backward ancestor list. If the next element is a child of the current one this is pushed into the stack; otherwise, the stack is used to locate its parent element, by comparing its DNN with the SPaR range of the elements in the stack. The algorithm is shown in Figure 4.

5 Support for Complex Queries

In addition to whole version reconstruction, other queries are important. For example, from version V_i, we may just want to retrieve the abstract, or the conclusion section, or the document segment from the fifth chapter till the

```
VersionReconstruction(SORTED_LIST) {
  Initialize ANCESTOR_STACK as empty;
  Assign the first element of SORTED_LIST as ROOT
    and remove it from SORTED_LIST;
  Push ROOT into ANCESTOR_STACK;
  current_node = ROOT;
  For (each element,E,in SORTED_LIST from the beginning)
  {
    if (SPaR(current_node) contains SPaR(E))
      Insert E as the first direct child of current_node;
      Push E into the ANCESTOR_STACK;
    else {
      do {
        Pop the top element, A, from ANCESTOR_STACK and
          compare SPaR(A) with SPaR(E);
        if (SPaR(A) contains SPaR(E)) {
         Insert E as the next direct child of A;
         Push A back into ANCESTOR_STACK;
         Push E into the path_stack
        }
      } while (the parent of A is found);
    }
    Set E as the current_node for the next run;
  }
}
```

Figure 4. Version reconstruction algorithm.

tenth chapter. Similarly, we may need the second through the sixth subsections under the fourth section of chapter ten in version V_i. A complete path in the document tree is provided in all these queries. Yet, many other useful queries use a regular expression to specify a pattern for the path, rather than giving the complete path. For example, an expression such as `version[i]/chapter[10]/*/figure` might be used to find all figures in chapter 10 of version V_i (or, symmetrically, the chapter that contains a given figure in version V_i). To support these queries efficiently, additional indices are needed, as discussed next. Since we assumed that the actual element records are organized in data pages using the page usefulness clustering scheme, these new indices will be dense: for each indexed element, a pointer to the element's position in the data pages is maintained. (In contrast, the Snapshot Index is a sparse index.)

Main Document Index. Consider a B+-tree indexing the element DNNs in the first version of a document. Each element is stored in this B+-tree as a record that contains the element ID, tag, SPaR DNN (and range) as well as a pointer to the data page that contains the actual data of this element. Such B+-tree facilitates interesting queries on the document's first version. For example, if we know the SPaR range of chapter10 we can find all document elements in this chapter (a range search). As the document evolves through versions, new elements are added, updated or deleted from this list. These changes can update the above B+-tree using the element DNNs. In order to answer SPaR range queries over a multiversion document (for example: 'find the elements in version V_i with DNNs in range (x, y)'), we need to maintain the multiple versions of this B+-tree.

Various multiversion B+-tree structures have been proposed [12, 1, 20]. Here we use the Multiversion B-tree (MVBT) [1] which has optimal asymptotic behavior and its code was readily available. The MVBT has the form of a directed graph with multiple roots. Associated with each root is a consecutive version interval. A root provides access to the portion of the structure valid for the root's version interval. While conceptually the MVBT maintains all versions of a given B+-tree, it does not store snapshots of these versions, since this would require large space. Instead, portions of the tree that remain unchanged through versions are shared between many versions. The MVBT uses also a notion of page usefulness: records that are alive for many versions are copied when a page gets below a usefulness threshold. However, the page splits are more elaborate than in the Snapshot Index, since the MVBT maintains also the order among all elements alive in a given version. As a result, each new page in the MVBT needs to allocate some free space for future elements that may be added in this page.

For a B+-tree evolution with n changes, the MVBT uses $O(n/B)$ space. Updating an element takes $O(log_B\, n_i/B)$, where n_i is the number of elements in the current version. A query requesting the elements in range R from version V_i is answered in $O(log_B n/B + r/B\,)$ I/O's, where r is the number of elements version V_i had in range R.

We use the MVBT to index all element records of the document; each such record points to the page that contains the element's data. In the rest we will refer to this index as *all_MVBT*. The index assumes that the SPaR DNNs are available. However, SPaR numbers are invisible to the user who expresses queries in terms of document tag names (abstract, chapter, etc.). Therefore, given a tag and a version number, the DNN of this tag in the given version must be identified.

We classify the document tags as either *list* tags or *individual* tags. Individual tags can only occur a small number of times in the document—typically only once, such as the abstract and the conclusion section. In some cases, individual tags occur a few times in a document (e.g., we might have an address tag for both sender and receiver). However, list tags, such as chapters, sections (and all the tags under them), can occur an *unlimited number of times* in a document. For individual tags, the SPaR number information can be easily maintained and accessed. Consider for example a query asking for the abstract in version 10. Assume that under the abstract tag, the document contains a subtree that maintains the abstract text, and a list of index terms. While the abstract SPaR remained unchanged, the subtree under the abstract tag may have changed. That is, the abstract text and the index terms could have changed between versions. To answer the above query we simply perform a range search (using the abstract's SPaR range) on the *all_MVBT* for version 10.

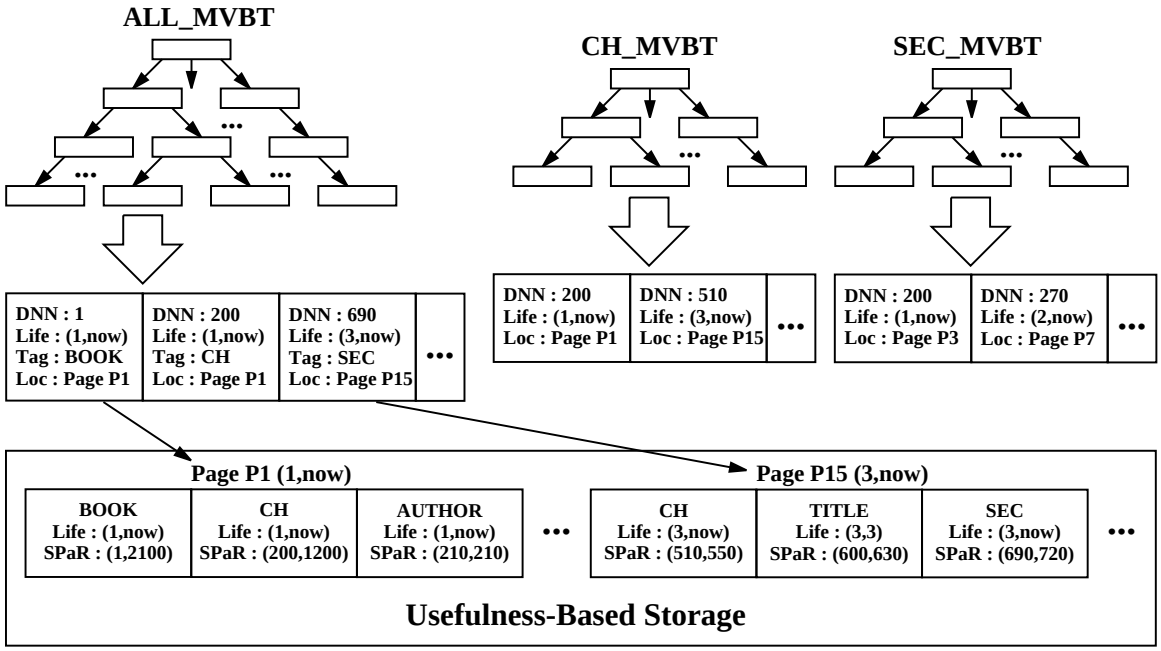

Figure 5. Dense element index structure.

Element Indices. Determining the SPaR numbers of list tags is more complex. This is because a new tag added in the list affects the position of all tags that follow it. For example, adding a new chapter after the first chapter in a document, makes the previously second, third,.., chapters to become third, fourth etc. Hence to identify the DNN of the tenth chapter in version 20, we need to maintain the ordered list of chapter DNNs. Such a list can also be maintained using a MVBT that indexes the SPaR DNN of $chapter$ tags (the *ch_MVBT*). We also maintain one MVBT per list tag in the document (for example, *sec_MVBT* indexes all document sections while *fig_MVBT* indexes all figures.) The dense index architecture is illustrated in Figure 5. At the bottom level, the data pages organized by the usefulness page clustering are shown (the structures of the Snapshot Index is not shown). Here each page has a usefulness interval and contains three element records. Each record has its tag, SPaR range, lifespan and data (not shown). Records in MVBT leaf pages contain pointers to the data pages. An MVBT leaf page contains more records than a data page since the latter stores the element data as well.

Below we provide various query examples and describe how to use the various indices in answering them efficiently. This list is a representative of the various queries that can be easily addressed through our index organization.

Structural Projection — *Project the part of the document between the second and the fifth chapters in version 20.* To answer this query we first access the *ch_MVBT* and retrieve the ordered list of chapter DNNs as it was in version 20. From this list we identify the SPaR range between chapters 2 and 5. With this SPaR range we perform a range search for version 20 in the *all_MVBT*. This search will identify all elements with DNNs inside this range. From the SPaR properties, all such elements are between chapters 2 and 5.

Regular Path Expression— *Find all sections under the third chapter in version 10.* We first identify the SPaR range of the third chapter in version 10 from *ch_MVBT*. With this SPaR range we perform a range search in the *sec_MVBT* for version 10. Only the sections under the third chapter will have SPaR numbers in the given range.

As another example, consider the query: *find the chapter that contains figure 10 in version 5.* To answer this query we first identify the DNN of the tenth figure in version 5 from *fig_MVBT*. Using this SPaR we perform a search in *ch_MVBT* for version 5. According to the properties of the SPaR numbering scheme, we find the chapter with the largest SPaR among the chapters before the figure, and check that its SPaR range contains that of the figure.

Parent-Child Expression— *For version 10, retrieve all titles directly under chapter elements.* Using the *ch_MVBT* we identify the $chapter$ elements alive in version 10. For each $chapter$, its SPaR range value is used to locate all $title$ elements under it in version 10 through the *title_MVBT*. Then, the level number of located titles are compared with

that of the chapter element to determine their parent-child relationship.

Version Selection— *Retrieve versions between 10 and 15.* The full version reconstruction algorithm can be used for version-ranges. The algorithm will start from version 15 and work backwards until version 10.

Query on Diff— *Retrieve the difference between versions* V_M *and* V_{M+1}. Basically, the difference is the union of inserted elements and deleted elements. Inserted elements are those elements whose creation time is version V_{M+1} and deleted elements are those elements whose lifespan ends by V_M. Version queries with predicates on the element lifespans can be used to retrieve these two types of elements.

6 Performance Results

We report preliminary experimental results on the use of the usefulness-based clustering. We used a document evolution with 100 versions. For simplicity, in this evolution each version has approximately the same size (about 100 pages). Each version changes about 20% from the previous version; half of the changes are insertions and the other half are deletions. Changes are uniformly and randomly distributed among data pages. The U_{min} is set to 50% while the page size is 4K bytes. We compare the SPaR data organization, the UBCC edit-script approach [4], the RCS and an extreme approach that stores the full copy of each version. The SPaR and UBCC both use the usefulness-based clustering. The difference is that UBCC relies on the edit script for identifying the useful pages while the SPaR uses timestamps and the access forest. For each method we observed the full version retrieval cost and the space consumption (Figure 6).

The retrieval cost is measured as the number of page I/O's needed to reconstruct a full version. Obviously, the approach that stores complete versions has the minimum version retrieval cost, but also the maximum storage cost, since each version is stored in its entirety on disk. Symmetrically, the RCS scheme requires the least storage (no usefulness based copying is performed) but has the largest average retrieval cost. The usefulness-based schemes (SPaR and UBCC) trade-off between these two extremes. The average retrieval cost for the usefulness-based schemes remains linear in the size of the reconstructed version by a coefficient that is controlled by U_{min}. Since in this experiment, the average version size was kept to about 100 pages and $U_{min} = 50\%$ the retrieval cost is approximately parallel to the horizontal axis (between 130 and 150 pages).

When comparing SPaR and UBCC, the SPaR scheme provides some improvement in retrieval time (about 10%) and more improvement in storage. The main reason is that the SPaR scheme does not need the edit script. Since the access forest uses one record per data page used, its structure is rather small when compared with the edit script. Nevertheless, the UBCC scheme is more robust since it does not need DNNs.

7 Conclusions

Versioning schemes for XML documents play an important role in the management of web based information. Traditional text versioning techniques such as RCS and SCCS are not efficient for XML document versioning. Hence there is a need for new and improved techniques that are more effective at the physical level and the logical level.

For the physical level, we have used a page clustering technique from temporal databases to trade off storage efficiency with retrieval efficiency and optimize the overall performance. Then we have concentrated on support for complex queries including queries on version content, queries involving the structure of the XML document (e.g., path expression queries), and temporal queries on the evolution of the document. We used a durable node numbering scheme that combined with indexing techniques, such as multiversion B-trees, can support well these three kinds of queries. Other interesting problems, such as the optimization of queries that combine these three different kinds of qualifications, performance comparison of alternative versioning schemes, and generalization of XML query languages to determine document evolution, will be the topic of future investigation. Indeed the problem of managing and querying multiversion XML documents presents many interesting research challenges for databases and web-based information systems of the future.

References

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

[2] D. Chamberlin, J. Robie, D. Florescu, *"Quilt: An XML Query Language for Heterogeneous Data Sources"*, The Proc. of the Third International Workshop on the Web and Databases, WebDB 2000, Dallas, Texas, May 2000.

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

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

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

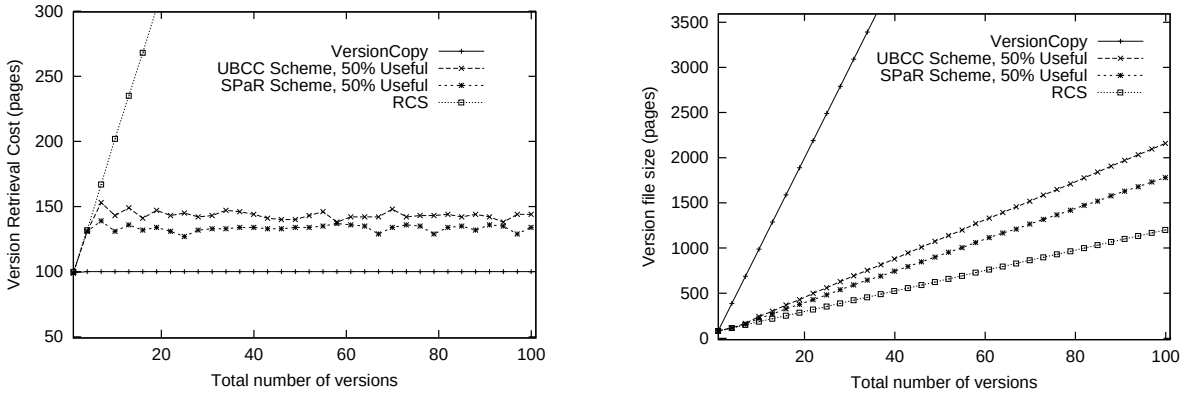

Figure 6. Version retrieval and storage cost with 50% usefulness requirement.

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

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

[8] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, R. T. Snodgrass, *"On the Semantics of "Now" in Databases"*. TODS 22(2): 171-214 (1997)

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

[10] N. Koudas, personal communication, 2001. D. Srivastava, S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, Y.Wu, *"Structural Joins: A Primitive for Efficient XML Query Pattern Matching"*. To appear, Proc. ICDE Conf., San Jose, CA, 2002.

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

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

[13] 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.

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

[15] R.H. Katz, E. Chang, *"Managing Change in Computer-Aided Design Databases"*, Proc. of the International Conf. on Very Large Databases (VLDB), 1987.

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

[17] J. Shanmugasundaram, et al, *"Relational Databases for Querying XML Documents: Limitations and Opportunities"* The Proc. of the 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK, September, 1999.

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

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

[20] 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.

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

[22] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. *"The Design and Performance Evaluation of Various XML Storage Strategies"*, http://www.cs.wisc.edu/niagara/Publications.html

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

[24] 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.