Efficient Complex Query Support for Multiversion XML Documents*

Shu-Yao Chien[1], Vassilis J. Tsotras[2], Carlo Zaniolo[1], and Donghui Zhang[2]

[1] Department of Computer Science, University of California, Los Angeles, CA 90095
{csy,zaniolo}@cs.ucla.edu
[2] Computer Science Department, University of California, Riverside, CA 92521
{tsotras,donghui}@cs.ucr.edu

Abstract. Managing multiple versions of XML documents represents a critical requirement for many applications. Also, there has been much recent interest in supporting complex queries on XML data (e.g., regular path expressions, structural projections, DIFF queries). In this paper, we examine the problem of supporting efficiently complex queries on multiversioned XML documents. Our approach relies on a scheme based on durable node numbers (DNNs) that preserve the order among the XML tree nodes and are invariant with respect to updates. Using the document's DNNs various complex queries are reduced to combinations of *partial version retrieval* queries. We examine three indexing schemes to efficiently evaluate partial version retrieval queries in this environment. A thorough performance analysis is then presented to reveal the advantages of each scheme.

1 Introduction

The management of multiple versions of XML documents finds important applications [28] 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

* This work was partially supported by NSF grants IIS-0070135, IIS-9907477, EIA-9983445, and the Department of Defense.

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 web sites 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 [18].

Various techniques for versioning have also been proposed by database researchers who have focused on problems such as transaction-time management of temporal databases [19], support for versions of CAD artifacts in O-O databases [12] and, more recently, change management for semistructured information [7].

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:

- 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 [22, 24] and the emergence of powerful XML query languages [2, 11, 5, 29, 6, 30]. A particularly challenging problem is that of supporting efficiently path expression queries such as:

`doc/chapter/ * /figure`.

This query specifies figures that are immediate elements of chapters or their transitive sub-elements (e.g., figures in sub-sections). Various techniques have been proposed to support regular path expressions [14–16] in the literature. These techniques use durable numbering schemes to preserve the logical document structure in the presence of updates.

For multiversion documents, we have to support such complex queries on any user-selected version. Furthermore, we need to support difference queries between two versions, and queries on the evolution of documents or selected parts of it, such as for lineage queries.

In [8] and [9] we proposed schemes for the efficient storage and retrieval of multiversion documents and showed that these provide significant improvements with respect to traditional schemes such as RCS [23] and SCCS [20]. To enhance the version retrieval efficiency, [8] places document elements in disk pages using a clustering mechanism called UBCC (for Usefulness Based Copy Control).

The UBCC mechanism achieves better version clustering by copying elements that live through many versions. A variation of UBCC was used in [9], where a reference-based versioning scheme was presented.

While the versioning schemes proposed in [8, 9] are effective at supporting simple queries, they cannot handle complex queries such as the path-expression queries. For complex queries, we have recently outlined [10] the *SPaR* scheme that adapts the *durable node numbers* [16] to a multiversion environment. Furthermore, SPaR uses *timestamping* to preserve the logical structure of the document and represent the history of its evolution. In this paper, we expand the properties of the *SPaR* scheme and investigate efficient physical realizations for it. Different storage and indexing strategies are examined so as to optimize SPaR's implementation. Our study builds on the observation that evaluating complex version queries mainly depends on the efficiency of evaluating one basic type of query: the *partial version retrieval* query. Such query retrieves a specific segment of an individual version instead of the whole version. Retrieving a segment for a single-versioned XML document is efficient since the target elements are clustered on secondary store by their logical order, but this might not be the case for a multiversion document. For a multiversion document, a segment of a later version may have its elements physically scattered in different pages due to version updates. Therefore, retrieving a small segment could require reading a lot of unnecessary data.

While UBCC is very effective at supporting full version retrieval queries, complex queries on content and history combined call for indexing techniques such are the Multiversion B-Tree [17, 3, 27] and the Multiversion R-Tree [13]. We investigate the following three approaches:

Scheme 1: single Multiversion B-Tree,
Scheme 2: UBCC with a Multiversion B-Tree, and
Scheme 3: UBCC with a Multiversion R-Tree.

The last two approaches still use the UBCC mechanism as the main storage scheme for the document elements. The additional indices are used as secondary indices so that partial version retrievals are efficiently supported. The first approach lets the Multiversion B-Tree organize the document elements in disk pages and at the same time uses the index for partial retrievals. The Multiversion B-Tree also uses a clustering technique. However, this technique is more elaborate and uses more disk space, since it clusters by versions and (durable) element numbers. A performance evaluation is presented to compare the the different schemes.

The rest of this paper is organized as follows. Section 2 provides background, while section 3 presents the SPaR scheme. In section 4, the three storage and indexing combinations are described. Their performance is presented in section 5 while conclusions appear in section 6.

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) edit script. Such 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 is obtained by applying, to the pair (V_j, V_{j+1}), a structured DIFF package [4].

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 SCCS 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 [8] we introduced an edit-based versioning scheme that (i) separates the actual document data from the edit script, and (ii) uses the usefulness-based clustering scheme (UBCC) 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 [17, 25, 3] to cluster temporal data and is outlined below.

2.1 Usefulness-based clustering

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. 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 are marked 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 retrieval 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* diminish. We would like to maintain a minimum page usefulness, U_{min}, over all versions. 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 (hence the name UBCC). When copied records are stored in new disk pages they preserve their relative document order. 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. The value of U_{min} is set between 0 and 1 and represents a performance tuning parameter.

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).

The advantage of UBCC is that the records valid for a given version are clustered into the disk pages that are useful for that version. Reconstructing the full document at version V_i is then reduced to retrieving only the pages that were useful at V_i. Various schemes can be used to assure that only useful pages are accessed for full version retrieval. For example [8] uses the edit script to determine the useful pages for each version, while [9] facilitates the object references.

While the UBCC clustering is very effective for full version retrieval queries, it is not efficient with complex queries like path-expression queries. Path-expression queries need to maintain the logical document order and UBCC does not.

2.2 Path-expression queries

A path-expression query is described by a regular expression on the document tree. For example, the query "find all the figures in chapter 10 of the document" is supported in XML query languages [6] 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 (in a single-version environment) without fully traversing document subtrees, a method is needed to quickly identify ancestor-descendant relationships between document elements. [16] proposes a numbering scheme for the document elements, whereby the numbers assigned to elements remain unchanged even if elements are added/deleted from the document. This is achieved by sorting the nodes as in the pre-order traversal, but leaving space between them to make room for future insertions.

Such a durable numbering scheme is advantageous since it automatically maintains the logical document tree structure. An ordered list with the node durable numbers is enough to reconstruct the document in logical order. Moreover since the numbering scheme does not change, it allows indexing various document elements in a persistent way. In [10] we outlined SPaR, a new versioning scheme that adapts the durable numbers of [16] as well as timestamps in order to efficiently support complex queries on multiversion XML documents. Below we expand the SPaR scheme properties and justify the reduction on various complex multiversion queries to partial version retrieval queries.

3 The SPaR Versioning Scheme

The new versioning scheme assigns durable structure-encoding ID numbers and timestamps to the elements of the document. SPaR stands for Sparse Preorder and Range, i.e., the numbering consists of two numbers: a Durable Node Number (DNN) and a Range, discussed next.

3.1 The Numbering Scheme

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 pre-order traversal number can then to identify uniquely the elements of the XML tree. While this 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 pre-order 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 [16]. Furthermore, these durable IDs must also describe the position of the element in the original document— a requirement not typically found for IDs in O-O databases. The DNN establishes the same total order on the elements of the document as the pre-order 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 [16] as a mechanism for supporting efficiently *path expression queries.* For instance, a document might have chapter elements and figure elements contained in such chapters. A typical query is: *"Retrieve all titles under chapter elements"*. Using recently proposed XML query languages ([6], etc.) this query is described as a path expression, as follows:

```
doc/chapter/ * /figure
```

In the XML document tree, figure elements could be children of chapter elements, or they might be descendants of chapter elements (e.g., contained in sections or subsections). To support the efficient execution of such path expression queries we need the ability of finding all the sub-elements of a given elements provided by the SPaR scheme. 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

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

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 $dnn(Y) + range(Y) \leq dnn(X) + range(X)$.

Thus, our insertion scheme assumes that an empty interval is at hand for every new element being inserted. When integers are used, occasional SPaR reassignments might be needed to assure this property. A better solution is to use floating point numbers, where additional decimal digits can be added as needed for new insertions. Nevertheless, for simplicity of exposition, in the following examples we will use integers.

Figure 1 shows a sample XML document with *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 1 and CH 2, and three insertion points, before CH 1, after CH 1 and after CH 2. 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.

3.2 The Version Model

Since the SPaR numbering scheme maintains the logical document order and supplies durable node IDs, it makes it possible to use timestamps to manage changes in both the content and the structure of documents. Hence the record of each XML document element contains the element's SPaR and the element's version 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 (if ever). An element is called "alive" for all versions

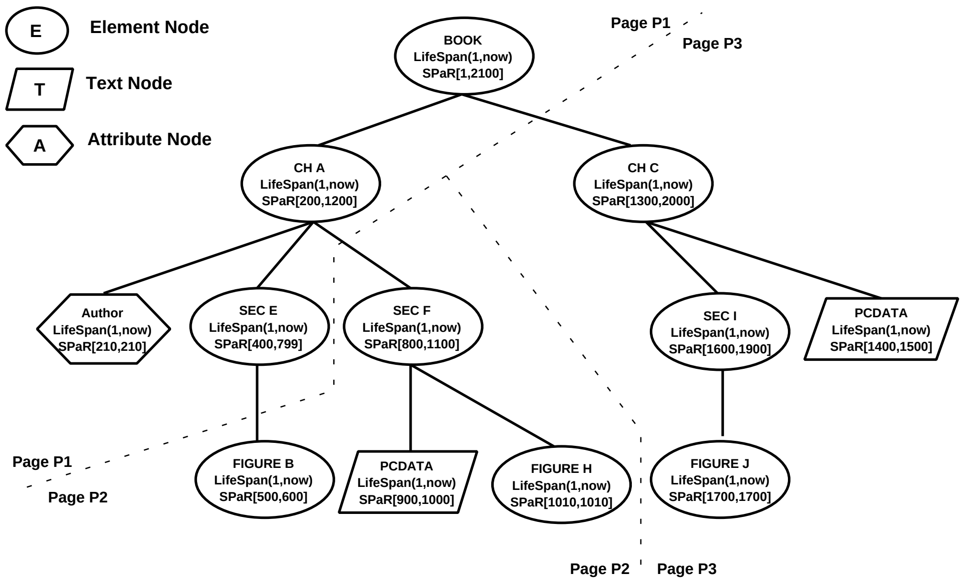

E
Element Node
T
Text Node
A
Attribute Node
BOOK LifeSpan(1,now) SPaR[1,2100]
CH A LifeSpan(1,now) SPaR[200,1200]
CH C LifeSpan(1,now) SPaR[1300,2000]
Author LifeSpan(1,now) SPaR[210,210]
SEC E LifeSpan(1,now) SPaR[400,799]
SEC F LifeSpan(1,now) SPaR[800,1100]
SEC I LifeSpan(1,now) SPaR[1600,1900]
PCDATA LifeSpan(1,now) SPaR[1400,1500]
FIGURE B LifeSpan(1,now) SPaR[500,600]
PCDATA LifeSpan(1,now) SPaR[900,1000]
FIGURE H LifeSpan(1,now) SPaR[1010,1010]
FIGURE J LifeSpan(1,now) SPaR[1700,1700]
Page P1
Page P3
Page P1
Page P2
Page P2
Page P3

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

in its lifespan. If an element is alive in the current version, its V_{end} value is *now* which is a variable representing the ever increasing current version number. A lifespan interval is left-closed and right-open; moreover, the lifespan of an element contains the lifespans of its descendants (much in the same way in which its SPaR interval contains those of its descendants). An example is shown in Figure 1.

The elements of the initial version, are stored as records in disk pages, ordered by their document order. In Figure 1 it was assumed that a page can hold four records (for simplicity all document elements have same size); the elements of the first version are stored in pages P1, P2 and P3, based on their SPaR order.

Successive versions are described as changes with respect to the previous version. Such changes are contained in an edit script generated from the structured XML editor, or otherwise by a package that computes the structured DIFF between the two documents. For simplicity we consider the following basic change operations: DELETE, INSERT and UPDATE. (Additional operations, such as MOVE or COPY elements can be reduced to these.) A new version is created by applying the basic operations on elements of the previous version. Below we discuss the effect of performing each basic operation, to create version V_N:

- *DELETE* — This operation updates the V_{end} timestamp of the deleted *element and all its descendants* from *now* to Version V_N. The *SPaR* range of the deleted elements is freed for reuse.
- *INSERT* — An INSERT operation creates a record for the newly inserted element and initializes its lifespan to (V_N, now). An unused range is assigned to the new element based on the weighted range allocation algorithm. The new record is stored in the acceptor page.

- *UPDATE* — The V_{end} timestamp of the updated element is changed to Version V_N. Subsequently, a new record is created with lifespan initialized to (V_N, now). This new record keeps the same $SPaR$ values as the original record (since the position of the updated element in the document did not change).

UBCC and Full Version Retrieval. Consider a SPaR scheme that adopts the UBCC clustering strategy (section 2.1) as its physical storage management. In addition to the above updates, records are copied due to the UBCC page usefulness threshold. We will now discuss how UBCC leads to fast full version reconstruction.

The first step of reconstructing a complete version is to identify the useful pages for the specified version. 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 document 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 at V_i is then equivalent to finding which pages have intervals that contain V_i. This problem has been solved in temporal databases [19] with an access method called the Snapshot Index [25, 21]. If there were k useful pages at V_i, they are located with $O(k)$ effort.

These useful pages contain all document elements at V_i, however, the elements may be stored out of their logical document order. Therefore, the second step is to sort the elements by their SPaR number. (It should be noted that versioning schemes not based on durable numbers [8, 9] use the edit script or object references to reconstruct the document order). A straightforward sort over all useful pages is not as efficient as it may require reading various useful pages many times. A better solution takes advantage of the partial order existing among the elements stored in a useful page and uses a one-pass sort-merge approach. Finally, using the SPaR numbers and a stack mechanism, the document tree structure is reconstructed.

In the next subsection we reduce various complex versioning queries to partial version retrievals. Since the UBCC does not maintain the SPaR order additional indexing schemes will be needed.

3.3 Complex Queries

While full version retrieval is important, many other versioning queries are of interest. For example, we might want to find only the abstract (or the conclusions section) that the document had at version V_i, or the part of the document from the fifth until the tenth chapter in version V_i. Similarly, we may need subsections two through six in the fourth section of chapter ten in version V_i. A common characteristic of the above queries is that a path in the document tree is provided. Yet, other interesting queries are those that instead of providing an exact path, they use a regular expression to specify a pattern for the 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 address these queries efficiently, additional indices are needed. Consider the set of all element DNNs (and their SPaR ranges) in the first document version. As the document evolves to a new version, this set evolves by adding/deleting DNNs to/from the set. Assume that an index is available which (i) indexes this version-evolving set and (ii) can answer retrieval queries of the form: "given a DNN range (x, y) and a version V_i, find which DNNs were in this range during version V_i". Since this index indexes all document elements, we will refer to it as the *full-index*.

The full-index assumes that the document 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.

Document tags are typically of two types. First, there are *individual* tags that only occur a small number of times in the document. Most of these tags, such as *example, abstract, references* and *conclusions* might occur only once in the document, although some individual tags can occur a few times (e.g., we might have an address tag for both sender and receiver). Then, there are *list* tags, such as: *chapters, sections*, and all the tags under them. Such tags can occur an unlimited number of times in a document. For simplicity assume that individual tags have a SPaR DNN and range that remain unchanged over versions. This information can be stored and accessed on request.

Consider for example a query requesting 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 SPaRs 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 full-index for version 10. 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 been maintained by using an index on the SPaR DNNs (and SPaR ranges) of *chapter* tags (the *ch_index*). Similarly with the full-index the ch-index can answer partial version retrieval queries specified by a version number and a range of chapter DNNs. We also maintain one index per list tag in the document (for example, *sec_index* indexes the DNNs of all document sections while *fig_index* indexes all figure DNNs.)

The overall index architecture is illustrated in Figure 2. This figure assumes that at the bottom level, the disk pages are organized by UBCC. Each UBCC page has a usefulness interval and contains three element records. Each UBCC record has its tag, SPaR range, lifespan and data (not shown). The records in the

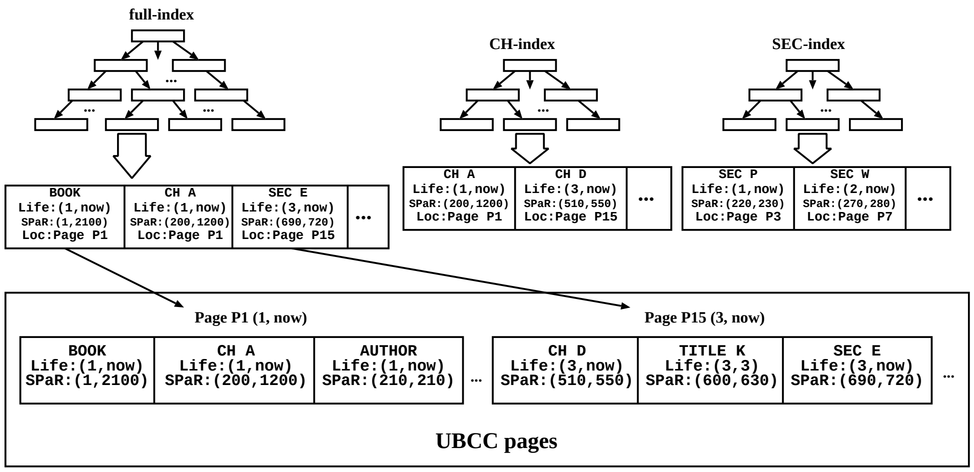

full-index
CH-index
SEC-index
BOOK Life:(1,now) SPaR:(1,2100) Loc:Page P1
CH A Life:(1,now) SPaR:(200,1200) Loc:Page P1
SEC E Life:(3,now) SPaR:(690,720) Loc:Page P15
CH A Life:(1,now) SPaR:(200,1200) Loc:Page P1
CH D Life:(3,now) SPaR:(510,550) Loc:Page P15
SEC P Life:(1,now) SPaR:(220,230) Loc:Page P3
SEC W Life:(2,now) SPaR:(270,280) Loc:Page P7
Page P1 (1, now)
Page P15 (3, now)
BOOK Life:(1,now) SPaR:(1,2100)
CH A Life:(1,now) SPaR:(200,1200)
AUTHOR Life:(1,now) SPaR:(210,210)
CH D Life:(3,now) SPaR:(510,550)
TITLE K Life:(3,3) SPaR:(600,630)
SEC E Life:(3,now) SPaR:(690,720)
UBCC pages

Fig. 2. The overall index architecture.

leaf pages of the full- and tag- indices contain pointers to the UBCC pages that contain these records. An index leaf page contains more records than a UBCC page since the latter stores the element data as well.

Using the above index combination various complex queries can be answered efficiently. These queries are first translated into partial version retrieval queries as the following examples indicate.

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_index* 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 *full_index*. 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_index*. With this SPaR range we perform a range search in the *sec_index* 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_index*. Using this SPaR we perform a search in *ch_index* for version 5. According to the properties of the SPaR numbering scheme, we find the chapter with the largest SPaR that is less than the figure SPaR.

Parent-Child Expression— *For version 10, retrieve all titles directly under chapter elements.* Using the *ch_index* 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_index*. Then, the level number of located titles are compared with that of the chapter element to determine their parent-child relationship.

4 Indexing Schemes

In this section we elaborate on the various data storage organization and indexing possibilities.

Data Storage Organization. We have already discussed one approach for the data storage organization, namely, UBCC. Another approach is to cluster the document element records using a multiversion B+-tree instead. 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 the actual data (text, image, etc) of this element. This 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). Furthermore, the full document can be reconstructed by simply following the leaf pages of this tree. As the document evolves through versions, new elements are added, updated or deleted. These changes can update the above B+-tree using the element DNNs. In order to answer queries over a multiversion document we need to maintain the multiple versions of this B+-tree.

Various multiversion B+-tree structures have been proposed [17, 3, 27]; here we consider the Multiversion B-tree (MVBT) [3] which has optimal asymptotic behavior for partial version retrievals and its code is 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 space that is proportional to the number of version changes. A query requesting the elements in range R from version V_i is answered with effort proportional to r, the number of elements version V_i had in range R.

Below we compare the choice of MVBT against UBCC for the data storage organization. An advantage of the MVBT is that it offers partial version retrievals, too. The MVBT also uses a notion of page usefulness. However, the page copies are more elaborate than in UBCC, since the MVBT maintains also the total order among all elements valid for a given version. As a result, each new page in the MVBT needs to preallocate some free space for future elements that may be added in this page. This implies that the space utilization of the MVBT is higher than UBCC. Moreover, the MVBT copies a page easier than the UBCC. This becomes important since the document element records are usually large (since they contain the element's data as well).

Full-index Implementation. The choice of the access method needed to implement the *full-index* is affected by the choice of the storage organization. If the

MVBT is used as the main storage organization, it serves as the full-index as well. Since the actual data records are stored at the leaves of this MVBT, it corresponds to a "primary" index. If the UBCC data organization is chosen, an additional index is needed so as to provide partial version retrievals. The leaf pages of this additional index will store element DNNs (and ranges) and pointers to the actual document element records in the UBCC pages. Hence it corresponds to a "secondary" index structure; furthermore, it is a dense index since all the element DNNs that exist in UBCC records appear also at the leaves of this index. As a result, a MVBT secondary dense index can be used to implement the full-index. Various optimizations on this combination of UBCC and MVBT can be applied as discussed in subsection 4.1.

We also propose an alternative that combines UBCC with a sparse secondary index, that indexes the range of element DNNs in each UBCC page. These ranges correspond to intervals and they may be updated as elements are added/updated in the page. To answer a range retrieval query, the index should identify the UBCC pages with range intervals that intersect the query range at the given version. Hence, this index must support multiversion interval intersection queries. The *Multiversion R-tree (MVRT)* [13, 26] is such an index. Like the MVBT, the MVRT conceptually maintains many versions of an ephemeral R-tree. We note that this MVRT is a sparse index: it does not store the element DNNs; rather, the ranges of page DNNs are stored. As a result, using the MVRT to implement the full-index will result in a much smaller structure.

Tag-index Implementation. A tag-index is used for each list tag, indexing the DNNs of all document elements using this tag. Since the actual element records are physically stored using the UBCC or the MVBT, each tag-index is a secondary dense index. To support partial version retrievals a MVBT or its variants can be used to implement a tag-index.

4.1 UBCC with dense secondary MVBT

Structures. When a new version is created, its updates are first applied on the document element records. Using these updates, the UBCC strategy may result into page copying. The alive records from the copied pages as well as the newly inserted document element records are first ordered by DNN. The ordered records are then placed in UBCC pages. Each update is also applied to the dense MVBT index. (A record copy is managed as the logical deletion of the previous record followed by a newly inserted record pointing to the record's new position in UBCC).

A partial version retrieval is accomplished by searching the MVBT using the given DNN range and version number. For all MVBT records found their pointers to the UBCC pages are followed to access the actual document element.

An interesting optimization is possible. First, note that in the above scheme, the version lifespan for a document element is kept in both the UBCC file and in the MVBT records. Second, when an element is updated as deleted the UBCC page that contains it is brought into main memory so as to change the record's lifespan. In the proposed optimization the version lifespans are kept in the MVBT index only. This saves space in the UBCC, and, saves I/O during element updates since the UBCC pages do not need to be accessed.

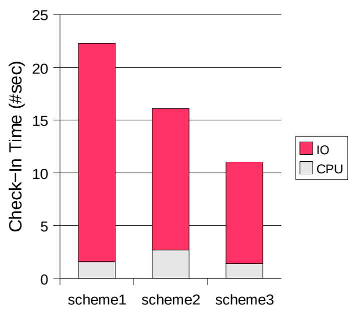
Nevertheless, when a UBCC page becomes useless in the optimized scheme, we need to know which of its records are still alive. This is implemented by a (main-memory) hashing scheme that stores the UBCC pages that are currently useful. The scheme is implemented on the page ids. For each useful page, the hashing scheme stores a record that contains: (UBCC page-id, current page usefulness, list of alive DNNs). The "current page usefulness" is a counter that maintains the number of alive records in the page. The "list of alive DNNs" lists the DNNs of these alive records. This hashing scheme is easily maintainable as records are updated/inserted.

4.2 UBCC with sparse secondary MVRT

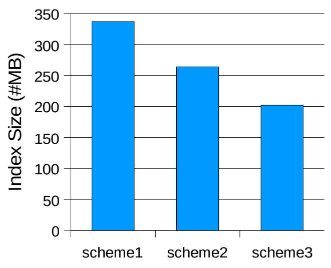
A difference from the previous approach is that the MVRT is used as a sparse index. Hence, the element lifespans are kept in the UBCC records. Moreover, to facilitate fast updates a hashing scheme that stores the currently alive document elements is implemented. For each alive element, the hashing scheme stores a record: (DNN, UBCC page-id), where the page-id corresponds to the UBCC page that stores this element. Element updates are then processed by first consulting this hashing scheme.

When a UBCC page becomes full, its DNN range is computed and inserted in the MVRT. In particular, the MVRT record contains: (DNN range, lifespan, UBCC page-id). This range is the largest DNN range this page will ever have since no new records can be added in it. While records are logically deleted from this page its DNN range may decrease. However, to save update processing, the MVRT stores the largest DNN range for every page. When a UBCC page becomes useless, the MVRT updates the lifespan of this page's record. This update process may result in accessing a page which intersects the query DNN range but contains no alive element for the query version. However, in our experimental performance the savings in update were very drastic to justify few irrelevant page accesses at query time.

Special attention is needed when reporting the answer to a partial version retrieval. In particular, elements in the query range must be reported in DNN order. One straightforward approach is to find all elements in the query answer and sort them. A better approach is to utilize the fact that data pages created in the same version have their elements in relative DNN order (since new elements and coped elements are first sorted before stored in UBCC pages). Hence the following sort-merge approach is possible: (1) use the records retrieved from the MVRT to group the UBCC page references by the V_{start} version in their lifespan, then (2) treat each group of data pages as a sorted list of objects and merge them using a standard sort-merge algorithm. With enough memory buffer a single scan of the data pages is sufficient.



(a) Check-In time



(b) Index sizes

Fig. 3. Comparing the check-in time and the index sizes

5 Performance

We present results comparing the performance of the three choices for data organization and full-index implementation. Scheme 1 uses a single dense primary MVBT, scheme 2 uses UBCC plus a dense secondary MVBT (section 4.1), while scheme 3 uses UBCC plus a sparse MVRT index (section 4.2). The usefulness in both UBCC based schemes was set to 0.5.

The schemes were implemented in C++ using GNU compilers. The programs were run on a Sun Enterprise 250 Server machine with two 300MHz UltraSPARC-II processors using Solaris 2.8. We used a 20GB Seagate disk. To compare the performance of the various algorithms we used the estimated running time. This estimate is commonly obtained by multiplying the number of I/Os by the average disk page read access time, and then adding the measured CPU time. The CPU cost was measured by adding the amounts of time spent in *user* and *system* mode as returned by the *getrusage* system call. We assume all disk I/Os are random which was counted as 10ms. We used a 16KB page size. For all the three schemes we used the LRU buffering where the buffer size was 100 pages.

We present results using a representative dataset. The representative dataset contains 1000 versions. The initial version has 10,000 objects, and an object occupies on average 200 bytes. Each later version was generated by performing 10% changes to the previous version, where about half of the changes were deletions and the other half were insertions. Following the *80/20 rule*, we assume that 80% of the changes took place in 20% of the document.

For the scheme that uses UBCC and the dense MVBT index, we implemented the optimized approach which keeps the element lifespans in the MVBT and utilizes the alive-page hashing (see section 4.1). This optimization led to a drastic 35% improvement in version check-in time when compared to the original approach. (The space improvement was less, around 1%, while the query times of both approaches were equivalent.)

Figure 3a compares the check-in time per version of the three schemes, while figure 3b compares the index sizes. The version check-in time measures the total

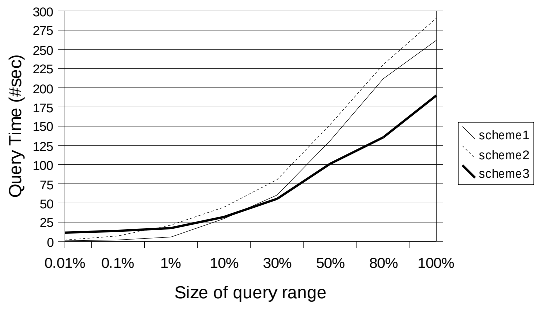


Fig. 4. Query performance varying length of DNN range.

time to finish all the updates within a given version. Scheme 3 has the fastest check-in time and uses the least index size. For the version check-in time, scheme 2 spent less I/O, but more CPU time than scheme 1. The MVBT used in scheme 1 stores the actual document elements in its leaf pages. Its update algorithms can trigger a copy more often than the UBCC. As a result, scheme 1 uses more update time and index space than scheme 2. Even though scheme 2 has also a secondary MVBT, this index is much smaller (since it does not contain the actual objects) and does not affect the relative performance. Scheme 3 is better than scheme 2 both in check-in time and in index size, since scheme 3 uses the sparse index (MVRT) which is much smaller than the dense index (MVBT) used in scheme 2.

To evaluate the performance of various queries, we measured the total execution time of 10 randomly generated queries with fixed-length DNN ranges. The performance is shown in figure 4. When the query DNN range is large, scheme 3 is the best. The reason is that a large query DNN range is like a full version retrieval, where all the UBCC pages that are useful at the query version need to be examined. In this case, scheme 3 is the most efficient since it spends the least time in finding these UBCC pages (as it uses a smaller sparse index). For small query DNN ranges, however, the other schemes are faster. The reason is that scheme 3 may check some UBCC pages which do not contain any qualifying object, while this is not the case for schemes 1 and 2. Scheme 1 is the best for small DNN ranges since it directly finds the qualifying objects. Scheme 2 has a little worse query time than scheme 1, since the actual objects and the references to these objects are maintained in two different structures and thus they have different clustering.

Overall, when considering both the check-in time, space as well as query time, the UBCC plus sparse MVRT (scheme 3) showed the most robust performance.

6 Conclusions and Future Work

As many applications make use of multiversion XML documents, the problem of managing repositories of such documents and supporting efficient queries on such repositories poses interesting challenges for database researchers. Foremost among these, is the efficient execution of complex queries, such as the path expression queries which are now part of the XML query languages being proposed as standards [29, 30]. In this paper, we investigated the problem of supporting these complex queries on multiversion documents. To solve this problem, we proposed solutions based on three main ideas:

- a durable numbering scheme for the document elements to capture their position and parent/child relations in a version-independent fashion,
- a version storage and clustering scheme based on the notion of page usefulness, and
- various multidimensional indexing schemes such multiversion B-trees and R-trees.

We first showed that, using this approach, complex path expression queries can be reduced to partial version retrieval queries; then, we evaluated alternative indexing and clustering schemes for the efficient execution of partial version retrieval queries. In addition to full version reconstruction, the proposed solution supports efficiently complex queries on version content, and queries involving the structure of the XML document (e.g., path expression queries).

Note that this paper assumes that there are no tag repetitions in the document tree paths. In that case, path queries involve join operations [16, 1]. We are currently investigating efficient ways to address multiversion path queries under tag repetitions. Other interesting problems, such as the generalization of XML query languages to determine document evolution and the optimization of such queries, will also be the topic of future investigations.

Acknowledgments

The authors would like to thank Bongki Moon for many useful discussions.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", *Proc. of ICDE*, 2002.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. L. Wiener, "The Lorel Query Language for Semistructured Data", *Journal on Digital Libraries* 1(1), pp. 68-88, Apr 1997.
3. B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree", *VLDB Journal* 5(4), pp. 264-275, 1996.
4. G. Cobena, S. Abiteboul and A. Marian, "XyDiff Tools Detecting changes in XML Documents", http://www-rocq.inria.fr/~cobena
5. S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi and L. Tanca, "XML-GL: A Graphical Language for Querying and Restructuring XML", *Proc. of WWW Conf.*, pp. 93-109, 1999.
6. D. Chamberlin, J. Robie, D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", *Proc. of WebDB*, 2000.

7. S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom, "Change Detection in Hierarchically Structured Information", *Proc. of SIGMOD*, 1996.
8. S.-Y. Chien, V.J. Tsotras and C. Zaniolo, "Version Management of XML Documents", *WebDB Workshop*, 2000.
9. S.-Y. Chien, V.J. Tsotras and C. Zaniolo, "Efficient Management of Multiversion Documents by Object Referencing", *Proc. of VLDB*, 2001.
10. S.-Y. Chien, V.J. Tsotras, C. Zaniolo, and D. Zhang, "Storing and Querying Multiversion XML Documents using Durable Node Numbers", *Proc. of WISE*, 2001.
11. A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu, "A Query Language for XML" , *Proc. of WWW Conf.*, pp. 77-91, 1999.
12. R. H. Katz and E. Change, "Managing Change in Computer-Aided Design Databases", *Proc. of VLDB*, 1987.
13. A. Kumar, V. J. Tsotras and C. Faloutsos, "Designing Access Methods for bitemporal Databases", *IEEE TKDE* 10(1), pp. 1-20, 1998.
14. M. Fernandez and D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas", *Proc. of ICDE*, 1998.
15. J. McHugh and J. Widom, "Query optimization for XML", *Proc. of VLDB*, 1999.
16. Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions", *Proc. of VLDB*, 2001.
17. D. Lomet and B. Salzberg, "Access Methods for Multiversion Data", *Proc. of SIGMOD*, pp. 315-324, 1989.
18. A. Marian, S. Abiteboul, G. Cobena and L. Mignet, "Change-Centric Management of Versions in An XML Warehouse", *Proc. of VLDB*, 2001.
19. G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey", *IEEE TKDE* 7(4), pp. 513-532, 1995.
20. M. J. Rochkind, "The Source Code Control System", *IEEE Tran. on Software Engineering* SE-1(4), pp. 364-370, Dec 1975.
21. B. Salzberg and V. J. Tsotras, "Comparison of Access Methods for Time-Evolving Data", *ACM Computing Surveys* 31(2), pp. 158-221, 1999.
22. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. J. DeWitt and J. F. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities" *Proc. of VLDB*, pp. 302-314, 1999.
23. W. F. Tichy, "RCS–A System for Version Control", *Software–Practice&Experience* 15(7), pp. 637-654, July 1985.
24. 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
25. V.J. Tsotras and N. Kangelaris, "The Snapshot Index: An I/O-Optimal Access Method for Timeslice Queries", *Information Systems* 20(3), pp. 237-260, 1995.
26. Y. Tao and D. Papadias, "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries", *Proc. of VLDB*, pp. 431-440, 2001.
27. P. Varman and R. Verma, "An Efficient Multiversion Access Structure", *IEEE TKDE* 9(3), pp. 391-409, 1997.
28. webdav, WWW Distributed Authoring and Versioning, last modified: Jul 31, 2001. http://www.ietf.org/ html.charters/webdav-charter.html
29. World Wide Web Consortium, "XML Path Language (XPath)", version 1.0, Nov 16, 1999. http:// www.w3.org/TR/xpath.html
30. World Wide Web Consortium, "XQuery 1.0: An XML Query Language", W3C Working Draft Jun 7, 2001 (work in progress). http://www.w3.org/TR/xquery/