

# Supporting Complex Queries on Multiversion XML Documents

SHU-YAO CHIEN

NCR/Teradata Division

VASSILIS J. TSOTRAS

CSE Dept., UC Riverside

CARLO ZANIOLO

CS Dept. UCLA

and

DONGHUI ZHANG

CCIS, Northeastern University

---

Managing multiple versions of XML documents represents a critical requirement for many applications. Recently, there has been much work on supporting complex queries on XML data (e.g., regular path expressions, structural projections, etc.). In this paper, we examine the problem of implementing efficiently such complex queries on multiversion XML documents. Our approach relies on a numbering scheme whereby durable node numbers (DNNs) are used to preserve the order among the nodes of the XML tree while remaining invariant with respect to updates. Using the document's DNNs, we show that many complex queries are reduced to combinations of *range version retrieval* queries. We thus examine three alternative storage organizations/indexing schemes to efficiently evaluate range version retrieval queries in this environment. A thorough performance analysis is then presented to reveal the advantages of each scheme.

Categories and Subject Descriptors: H.4.0 [Information Systems Applications]: General

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: XML document, multiversion, query support, version retrieval

---

## 1. INTRODUCTION

The management of multiple versions of XML documents finds important applications [Webdav 2001] 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 rep-

---

Communicating author's address: Donghui Zhang, CCIS, Northeastern Univ., Boston, MA 02115, donghui@ccs.neu.edu, phone: 1-617-373-2177, fax: 1-617-373-5121.

This work was partially supported by NSF grants IIS-0339032 and IIS-0339259.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2004 ACM 0000-0000/2004/0000-0001 \$5.00

resenting 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 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 [Marian et al. 2001].

The problem of version management has not originated with the web or XML; in fact it has been studied extensively in the context of document management systems and software configuration management, for which techniques such as RCS [Tichy 1985] and SCCS [Rochkind 1975] have been introduced. Various versioning techniques have also been proposed by database researchers who have focused on problems such as transaction-time management of temporal databases [Özsoyoglu and Snodgrass 1995], support for versions of CAD artifacts in O-O databases [Katz and Change 1987] and, more recently, change management for semi-structured information [Chawathe et al. 1996].

However in the past, database systems and document management systems were faced with different versioning problems since:

- 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 past state of affairs has now been changed 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 [Levy et al. 1997; Fernandez and Suciu 1998; McHugh and Widom 1999; Shanmugasundaram et al. 1999; Tian et al. 2002; Tatarinov et al. 2002] and the emergence of powerful XML query languages [Abiteboul et al. 1997; Deutsch et al. 1999; Ceri et al. 1999; WWW Consortium 1999; Chamberlin et al. 2000; WWW Consortium 2001]. Typically, an XML document is represented as an ordered tree, where nodes correspond to document elements. Query languages like [WWW Consortium 1999; 2001] provide for navigational queries through the document’s structure. A particularly challenging problem is that of supporting efficiently path-expression queries. Consider for example the query: “*Retrieve all*

*figures under chapter elements*”, which corresponds to:

`doc/chapter/*/figure.`

This query asks for **figures** that are immediate descendant elements of **chapter** or their transitive sub-elements; therefore, **figure** nodes may be anywhere in the subtrees rooted in `doc/chapter` nodes of the document. To answer such queries efficiently without fully traversing document subtrees, a method is needed to quickly identify ancestor-descendant relationships between document elements. To this end, numbering schemes have been proposed that capture the document tree structure. Given two tree nodes, their ancestor/descendant relationship can be found in constant time by comparing their respective numberings [Li and Moon 2001; Al-Khalifa et al. 2002; Cohen et al. 2002]. Moreover, *durable* numbering schemes have been also proposed [Li and Moon 2001; Cohen et al. 2002; Wang et al. 2003]. Such schemes have the advantage of being invariant to document updates.

In the case of multiversion documents, the above path-expression queries need be supported on any user-selected version, or even on a collection of subsequent versions. In summary, we are interested in addressing in a multiversion environment, complex queries like:

- *Structural Projection queries*, e.g. “project the part of the document between the second and the fifth chapters in version 20”. In this query, a continuous chunk of the document is requested, as it was at a given version.
- *Path-Expression queries*: given a regular path expression and a version, find the set of elements that satisfy the expression at the specified version. E.g. “find the figure elements in chapter 10 at version 5”. These queries differ from structural projections since document parts are selected based on a path-expression predicate. An important class of path expression queries are the *structural joins* [Al-Khalifa et al. 2002; Li and Moon 2001], e.g. “find all figures under chapters in version 20”. The binary join condition involves parent-child or ancestor-descendant structural relationships. Such structural relationships can be combined to form more complex “path” and “twig” joins.
- *Version-Interval queries*: The above, single-version queries, can be extended to multiple, subsequence versions (hence the name version-interval). For example, project a given part of the document as it was between versions 20 and 25, or, “find the figure elements within chapter 10 in versions 5 through 15”. Within this category, various other interesting queries exist. For example, a *Version Aggregation* query computes an aggregate value of elements specified by a version-interval query. E.g. “find the number of words in all text elements in chapter 10 in versions 5 through 15”. Similarly, a *DIFF* query returns the changes applied to a given object between versions e.g. “find the changes made to chapter 10 from version 5 until the current version”.

We should emphasize that addressing such queries is greatly facilitated by having captured the document’s structure directly through the numbering scheme. Previous XML versioning approaches [Chien et al. 2000; 2001; Marian et al. 2001] would be inefficient since they, respectively, use edit-scripts, references, and key identifiers to reconstruct the document’s logical order (i.e., no scheme exists to capture the

structure of the document ). In addition to the above queries, we will show that the numbering scheme can also facilitate *full-document version retrieval* queries, that is, reconstruct the full document as it was in a specific version.

We thus propose using a new versioning scheme (called *SPaR*, for Sparse Pre-order and Range) that adapts the *durable node numbers* of [Li and Moon 2001] to a multiversion environment. SPaR uses *timestamping* to preserve the logical structure of the document and represent the history of its evolution. Different storage and indexing strategies are examined so as to optimize SPaR's implementation. Our study builds on the observation that the efficiency of evaluating complex version queries mainly depends on the performance of a basic type of query: the *range version retrieval* query. Such query retrieves a specific segment of the document valid at an individual (or multiple subsequent versions). In the structural projection query above, we need to find among the whole document in version 20, only the segment (range) of interest. Many of the queries above correspond to one such range query over a single (or multiple versions) while structural (and path or twig) joins conceptually correspond to many range retrieval queries at the version(s) of interest.

Retrieving a segment (range) from a non-versioned XML document is efficient since the target elements are clustered on secondary store by their logical document 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.

To effectively support version-based queries, it is advantageous to cluster data according to their respective versions. We have recently introduced such a clustering mechanism [Chien et al. 2000] called *UBCC* (Usefulness Based Clustering Control) that achieves better version clustering by making copies of elements that live through many versions. While UBCC is very effective at supporting full version retrieval queries, complex queries on content and history combined call for indexing techniques that can support efficiently range version retrieval, like the Multiversion B-Tree [Lomet and Salzberg 1989; Becker et al. 1996; Varman and Verma 1997] and the Multiversion R-tree [Kumar et al. 1998; Tao and Papadias 2001]. Each technique implies a storage architecture for the multiversion system. We thus investigate the following three approaches:

*Scheme 1:*. using only Multiversion B-Trees,

*Scheme 2:*. combining UBCC with Multiversion B-Trees, and

*Scheme 3:*. combining UBCC with Multiversion R-trees.

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 to expedite range version retrieval. The first approach lets the Multiversion B-Tree organize the document elements in disk pages and at the same time uses the index for range version retrieval. The Multiversion B-Tree also uses version clustering. However, it clusters by versions and (durable) element numbers and typically uses more space. A performance evaluation is then presented to compare the different schemes.

The main contributions presented in this paper are summarized as follows:

- We propose techniques, based on the SPaR scheme, for reducing complex queries over multiversion XML documents to (single or combinations of) range version retrieval queries, and
- We present an in-depth study of the problem of implementing efficiently range version retrieval queries, and investigate the pros and cons of alternative approaches through extensive experiments.

This paper integrates and extends results presented in [Chien et al. 01b; 02b] in several ways discussed in the body of the paper. In addition to a comparative description of the methods and related work, we extend the techniques presented in [Chien et al. 01b; 02b] to support range deletions as well as incremental updates. Furthermore, we have enhanced the experimental evaluation to include performance comparisons for single-version and multiple-version range retrieval queries, as well as multiple range (join) version queries.

The rest of this paper is organized as follows. Section 2 provides related work, while section 3 presents the SPaR versioning scheme and discusses how full-version retrieval queries are addressed efficiently. Section 4 examines how various complex queries are reduced into combinations of range version retrievals. In section 5, the three storage and indexing combinations are described. Their performance is presented in section 6 while conclusions appear in section 7. The UBCC scheme is summarized in the Appendix.

## 2. RELATED WORK

*Path Expression Queries.* XML queries (e.g. [WWW Consortium 1999; 2001]) typically specify patterns of selection predicates on multiple elements that have some specified tree structure relationships. Path expressions represent the key component of such queries, and thus they are at the core of XML query processing. A path expression defines a series of XML tree node labels, which are related by ancestor-descendant or parent-child relationships.

Various works have recently addressed efficient implementation of such queries on non-versioned documents, and can be generally classified as “navigational” [Gottlob et al. 2002; 2003a; 2003b; Halverson et al. 2003; Wang et al. 2003; Diao et al. 2002; Rao and Moon 2004] and “set-based” [Al-Khalifa et al. 2002; Bruno et al. 2002; Chien et al. 02c; Halverson et al. 2003; Jiang et al. 2003; Li and Moon 2001; Wei et al. 2003; Jiang et al. 2003; Wei et al. 2003]. Navigational approaches analyze the input one tag at a time. On the other hand, the set-based approach views the input as a set and can thus use techniques from relational systems, like joins. Furthermore, this leads to an easier application of indexing solutions [Li and Moon 2001; Chien et al. 2002; Wei et al. 2003; Jiang et al. 2003].

Set-based approaches typically avoid traversing (navigating) through the document structure by embedding a numbering scheme [Al-Khalifa et al. 2002; Li and Moon 2001; Fiebig and Moerkotte 2000] on the XML tree nodes that capture their structural relationships. A commonly used numbering scheme is *range-based*, i.e., each node in the document tree is assigned a range of numbers that enables identifying the position of the node in the XML tree. Using range-based numbering, path-expression queries reduce to join operations [Li and Moon 2001; Al-Khalifa et al. 2002; Bruno et al. 2002]. [Li and Moon 2001] proposed a *durable* number-

ing scheme 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. Updatable, prefix-based numbering schemes have been also proposed [Cohen et al. 2002]; however, they use variable length encodings and typically need more space than range-based numbering. As a result, in this work we will utilize range-based numbering schemes.

*Version Management.* Versioning has been examined in CAD and O-O databases [Katz and Change 1987; Beech and Mahbod 1988] where the major concern was on version modeling. Two popular version management schemes have been developed for software configuration management [Leblang 1994], namely, RCS [Tichy 1985] and SCCS [Rochkind 1975].

RCS is edit-based: the most current document version is stored intact while previous versions are kept as *reverse* edit-scripts (changes). For any version except the current one, extra processing is needed to apply the reverse editing script and generate the old version. In a symmetric representation, SCCS stores the very first document version and maintains future versions with a *forward* edit script. Rather than appending version differences, SCCS inserts editing operations in the original document and associates a pair of timestamps (or version ids) with each document segment (object) to specify its lifespan. Versions are retrieved from an SCCS file via scanning through the file and retrieving valid segments based on their timestamps. Both schemes treat a document as a sequence of lines of text, and ignore the rich structure of documents, thus impairing the ability of supporting structural queries. Furthermore, they lack sophistication in their secondary storage management. Both RCS and SCCS may read document segments which are no longer valid for the retrieved (target) version, causing additional processing costs.

Previous work on version management of semi-structured documents [Chawathe and Garcia-Molina 1998] in general—and in particular of XML documents [Chien et al. 2001; Marian et al. 2001; Chien et al. 2002; Buneman et al. 2002; Wong and Lam 2002]—has considered various aspects of the storage and manipulation of multiversion documents. [Marian et al. 2001] considers queries on the XML document’s evolution. In [Chien et al. 2000; 2002] we proposed techniques that represent the document evolution as edit-scripts with the focus on fast reconstruction of any particular document version. To enhance the version retrieval efficiency, document elements are placed in disk pages by version, using the UBCC clustering mechanism, which achieves better version clustering by copying elements that live through many versions. Instead of using edit-scripts in [Chien et al. 2001] we presented a reference-based versioning scheme that is also optimized for full version retrieval queries. [Buneman et al. 2002] presents an approach based on SCCS that clusters historical information by object (i.e., all versions of the same object are physically clustered together). Moreover, its version querying assumes the existence of key identifiers.

It should be noted that the numbering scheme facilitated in this paper attains durability over updates by maintaining unused number ranges for future document elements. Such scheme used fixed sized numbers for labeling the document elements. There are also labeling schemes that are instead based on variable-length

labels, typically using prefixes [Abiteboul et al. 2001; Kaplan et al. 2002]. In particular, [Cohen et al. 2002] presents a durable variable length labeling scheme. We used range numbering mainly because of its ease in implementation; variable labeling may lead to labels with large length (in worse case proportional to the document depth). In [Vagena and Tsotras 2003] we have proposed a “region-based” numbering scheme for multiversion documents, where new updates create subsequence numbering regions while a structure is maintained to keep the relative sequence between regions.

Finally, the notion of linear versioning is similar to the notion of “transaction time” in temporal databases [Özsoyoglu and Snodgrass 1995]. A number of data structures has been proposed to index transaction-time databases [Salzberg and Tsotras 1999; Becker et al. 1996; Lomet and Salzberg 1989; Tsotras and Kangelaris 1995] and effectively maintain previous states (versions) of a database relation. Works have also addressed issues related to version branching [Lanka and Mays 1991; Landau et al. 1995; Jiang et al. 2000]. Recently, in [Vagena et al. 2004] we examine path expression queries in a branched multiversion documents.

### 3. THE SPaR VERSIONING SCHEME

The SPaR versioning scheme assigns durable structure-encoding ID numbers [Li and Moon 2001] and timestamps to the elements of the document. SPaR’s numbering consists of two numbers: a Durable Node Number (DNN) and a Range, discussed next. And as we will see, SPaR represents an interval  $[dnn(X), dnn(X)+range(X)]$ .

#### 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 uniquely identify 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 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. 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 is a mechanism that enables fast checking of ancestor/descendant relationships. 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  $A$ <sup>1</sup> iff:

<sup>1</sup>If the pre-order traversal number is used as DNN,  $range(A)$  is equal to the number of descendants of  $A$ .

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

Therefore, the interval  $[dnn(X), dnn(X) + range(X)]$  is associated with element  $X$ . Later on we use SPaR to represent this interval. When an element in the document is updated, the SPaR remain unchanged. When a new element is inserted, it is assigned a SPaR that does 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  $dnn(Y) + range(Y) \leq dnn(X) + range(X)$ .

The above insertions assume 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.

Fig. 1 shows a sample XML document with *SPaR* values. The root element is assigned a SPaR  $[1,2100]$ . That interval is split into five sub-intervals —  $[1,199]$ ,  $[200,1200]$ ,  $[1201,1299]$ ,  $[1300,2000]$ , 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 sub-interval, or SPaR, 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 Versioning 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. In the rest of the paper we

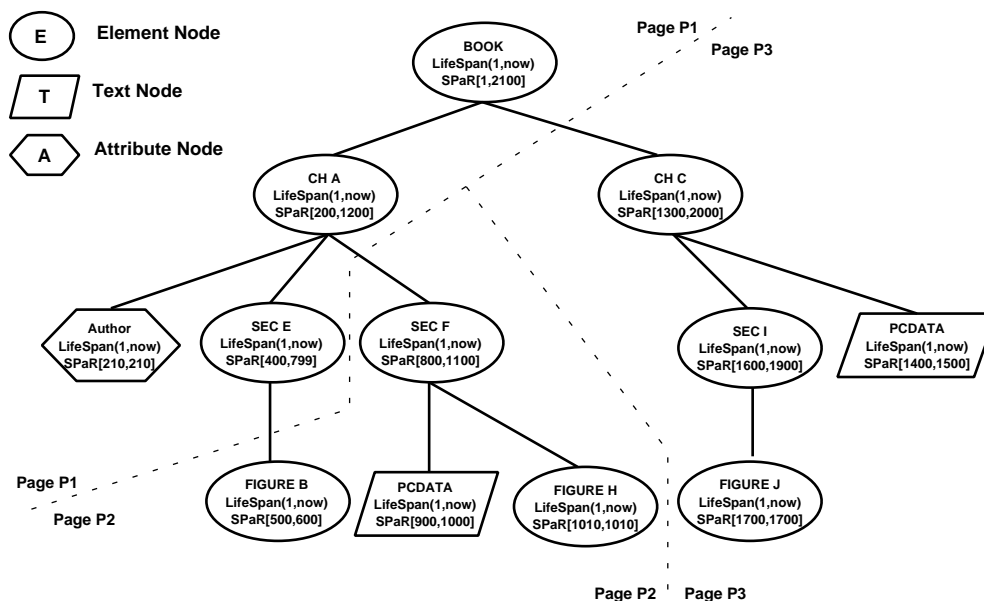


Fig. 1. An XML document version represented in the SPaR model.

assume the document follows *linear* versioning. That is, a new document version ( $V_{j+1}$ ) is established by applying a number of changes (object insertions, deletions or updates) to the current version ( $V_j$ ). As a result, document versions follow a linear order. 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 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 Fig. 1.

The elements of the initial version, are stored as records in disk pages, ordered by their document order. In Fig. 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  $V_N$ :

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

Fig. 2. The Page Repository for Version 2.

- DELETE*— This operation updates the  $V_{end}$  timestamp of the deleted *element and all its descendants* from *now* to Version  $V_N$ . The *SPaR* interval 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 interval is assigned to the new element based on the weighted interval 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).

### 3.3 UBCC and Full-Document Version Retrieval Queries

**Usefulness-Based Copying.** Consider a *SPaR* scheme that adopts the UBCC clustering strategy (see Appendix) as its physical storage management. In addition to the above updates, records are copied due to the UBCC page usefulness threshold.

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* 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.

**Full-Document Version Retrieval.** We will now discuss how UBCC combined with the SPaR numbering leads to fast full-version retrieval queries. 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. Each page  $p$  is thus represented by a record  $p_r$  that contains the page-id and the page's usefulness interval. 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 usefulness intervals that contain  $V_i$ . This problem has been solved in temporal databases [Özsoyoglu and Snodgrass 1995] with an access method called the Snapshot Index [Tsotras and Kangelaris 1995; Salzberg and Tsotras 1999]. If there were  $k$  useful pages at  $V_i$ , their  $p_r$  records (and thus their page-ids) are identified with  $O(k)$  effort.

*Ordering the Elements.* The pages identified by the above task, contain all document elements at  $V_i$ , however, these elements may be stored out of their logical document order among the useful pages. Therefore, the second step is to order the elements by their SPaR number. (It should be noted that versioning schemes not based on durable numbers [Chien et al. 2000; 2001] use the edit script or object references to reconstruct the document order).

A straightforward sort over all useful pages 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 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. The first (and in most cases the only) version that writes to page  $p$  (also called the page's *creator* version) is version  $V_{start}$  in  $p$ 's usefulness interval. Since a given version may write many pages, we need to retain the position  $p$  had among all pages written by  $p$ 's creator version. Such position can be maintained as an extra field within the  $p_r$  record of each page written by this version. For simplicity let us assume that a single version can write a page (this is easily implementable: the last page written by a version cannot be written by a future version even if it is not full; such an implementation does not affect usefulness, since the threshold applies to full pages). 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 page-ids created by that version and which correspond to useful pages for  $V_i$ . Such list can be built without physically accessing the useful pages in version  $V_i$  (since the position information is within the  $p_r$  records identified by the previous task).

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.

In case we allow a page to be written by more versions we need to maintain all such versions (in a form of a list). For example, the last page written by a given version may not be fully filled; a future version can then start adding elements in this page. Nevertheless, this page will be first among the pages that the future version writes. As such, this page will be among the first accessed by the sorting algorithm.

*Reconstructing the Document Structure.* To reconstruct the ordered-tree structure from an ordered list of elements 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 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 of the elements in the stack. The algorithm is shown in Figure 3.

#### 4. COMPLEX QUERY SUPPORT

While full document retrieval as of a particular version is important, many other versioning queries are of interest. In this section, we propose a storage scheme architecture and show how we can use this architecture to answer efficiently various complex versioning queries. As we will see, with the aid of the index architecture, we are able to reduce these queries to range version retrieval queries.

Assume that 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 these queries is that a path in the document tree is provided. Yet, other

```

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 child of current_node;
      Push E into ANCESTOR_STACK;
    }
    else
    {
      while (TRUE) // The stack contains some ancestor of E
      {
        Pop the top element, A, from ANCESTOR_STACK;
        if (SPaR(A) contains SPaR(E))
        {
          Insert E as the next child of A;
          Push A back into ANCESTOR_STACK;
          Push E into ANCESTOR_STACK;
          break;
        }
      }
      current_node = E;
    }
  }
}

```

Fig. 3. The Backward Ancestor Stack algorithm.

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  $V_i$ ).

Since the UBCC does not maintain the SPaR order, addressing these queries efficiently requires additional indices that maintain the SPaR order within each version. Consider the set of all element DNNs (and their 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  $(x, y)$  and a version  $V_i$ , find which DNNs were in  $(x, y)$  during version  $V_i$  (or between consecutive versions  $V_i$  and  $V_j$ )”. To answer such queries the index needs to maintain the order of element DNNs within each version. 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 few times in the document. Most of these tags, such as *abstract*, *references* and *conclusions* might occur only once in the document, although some

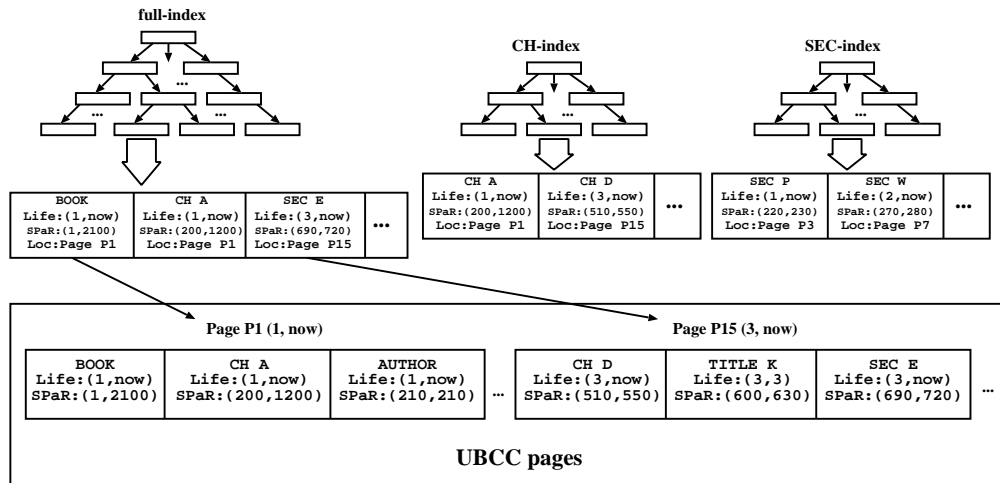


Fig. 4. The overall index architecture.

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) 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 by version. Such a list can also be maintained by using an index on the SPaR of *chapter* tags (the *CH-index*). Similarly with the full-index the *ch-index* can answer range 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 Fig. 4. 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, lifespan and data (not shown). The records in the 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 range 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 DNN range of elements in chapters 2 through 5. With it 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.

**Path Expression Queries** — *Find all sections under the third chapter in version 10.* We first identify the SPaR of the third chapter in version 10 from *CH\_index*. With this interval we perform a range search in the *SEC\_index* for version 10. Only the sections under the third chapter will have DNNs in the given interval.

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 DNN that is less than the figure’s DNN.

A special case of the Regular Path Expression query is the *Parent-Child Expression*, e.g. *For version 10, retrieve all titles “directly under” chapter 5.* We thus require that for every element inserted, the level number of the element in the XML tree is stored. To answer the above parent-child query, using the *CH\_index* we identify the *chapter 5* alive in version 10. The SPaR range of this *chapter* 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.

If instead the query asks: *For version 10, retrieve all titles directly under chapters,* we have an example of a “structural join”. Here, we first identify all *chapter* elements alive in version 10 from the *CH\_index* and all *title* elements in the same version. These elements are then joined using the containment property (as well as the parent-child relationship). It should be noted that the structures we are using to implement the tag indices effectively provide access to the elements of each tag list as of a given version, *ordered* by their DNN numbers. This is an important observation, since it allows adapting structural, path and twig join algorithms [Al-Khalifa et al. 2002; Li and Moon 2001; Bruno et al. 2002] to answer their version counterparts. For example, a structural join will be performed in a “merge-join” fashion, while each index can be facilitated to “skip” elements that will not join [Chien et al. 02c].

In a non-versioned environment, the stack-based algorithms in [Al-Khalifa et al. 2002] perform a structural join in one pass through the involved list tags. That is, the I/O for computing a structural join is proportional to the summation of the sizes of the involved lists. In a versioned environment, the use of a multiversion tag index, introduces a small overhead, due to the record copying that the index

maintenance and version clustering require. Consider performing a structural join that involves version  $V_i$  of a given list tag. Let  $L(i)$  denote the size of the list (in number of elements) as of version  $V_i$ . Thus a snapshot of the list at version  $V_i$  would occupy  $V_i/B$  pages ( $B$  is the page size in element records). It has been shown that such list can be maintained in the multiversion index with  $O(V_i/B)$  pages (i.e., linear overhead). For example, in our experiments, when a MVBT [Becker et al. 1996] is used as the multiversion index on a list tag, the average size of a versioned list was using about twice as many pages than the size of the list snapshot. In practice, a structural join would not traverse the whole list as of version  $V_i$  since the index can be used to *skip* list elements that are not involved in the join (in a similar way as in [Chien et al. 02c]).

**Version Interval Queries** — *Find the content of chapter 10 in versions 5 through 15.* This query is an extension in that we now consider *multiple* versions. To support such query, we first use the *CH\_index* to get the SPaR of chapter 10. Then we query the full index for elements whose DNNs are in the SPaR interval and whose versions are in the version interval [5, 10].

A special case of the Version Interval query is the following **DIFF query**, e.g. *find the changes made to chapter 10 from version 5 through version 8.* To support this query, we can perform a version interval query, while discarding the elements alive on or before version 5 on-the-fly.

Consider next the *Version Aggregation Query: Find the total number of figures in chapters 10 through 15 from version 5 through version 8. Or, find the total number of words in the text elements in chapters 10 through 15 within these versions.* These queries are not interested in the elements themselves, but in some aggregate value of these elements. The first example uses COUNT as the aggregation function, while in the second example uses SUM. That is, every text element corresponds to a value (the number of words in the text), and we are interested in the total value of selected elements.

One approach for solving the aggregation query is to perform a version-interval query to find the specified elements, and aggregate their values on-the-fly. However, the cost of this solution is proportional to the number of involved elements (which may be large). A better approach is to utilize specialized aggregation structures, like the Multi-version SB-tree [Zhang et al. 2001], which achieves logarithmic query time. This structure supports SUM, AVG, and COUNT queries in  $O(\log_b n)$  I/Os, where  $b$  is the fan-out of the tree and  $n$  is the number of elements. Furthermore, the index is dynamically updatable (an update takes  $O(\log_b K)$  I/Os, where  $K \leq n$  is the number of elements in the current version. To implement this idea, in the SUM example above, we maintain a Multi-version SB-tree on the text elements. Each record in the index has the format:  $\langle DNN, creation\_version, deletion\_version, num\_words \rangle$ . To perform the SUM aggregation query, we first search the *CH\_index* to retrieve the DNN range for chapters 10 through 15. Then, using this DNN range and the time interval [5, *now*], we query the Multi-version SB-tree to get the SUM of the number of words in the involved text elements.

Note that the Multi-version SB-tree is an optional, specialized index which can be added to improve aggregation queries. It is thus not considered further among

the basic storage organizations discussed next.

## 5. INDEXING SCHEMES

In this section, we study various data storage organizations and indexing alternatives. We first discuss briefly the implementation of a tag-index, which because of its small size has only a minor impact on performance, and then we use the rest of the section to investigate alternative implementations for the full-index.

For each list tag, we create a tag-index to index the DNNs of all document elements using this tag. Since the actual element records are physically stored somewhere else (e.g. using the UBCC as shown in Fig. 4), each tag-index is a small secondary index. To support range version retrieval (in single or multiple versions), an MVBT or its variants can be used to implement a tag-index.

### 5.1 Indexing Scheme One: using an MVBT

The architecture of Fig. 4 utilizes UBCC to cluster the data objects and uses a separate full-index which has a pointer to the data page of each indexed data object. This corresponds to scheme 2 in section 5.2. It is not the only choice. Since the primary interest here is to cluster data objects on their version and their DNN, and to support range version retrieval queries, we can utilize an Multi-version B+-tree to replace both the full-index and the UBCC.

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 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 of chapter 10 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 [Lomet and Salzberg 1989; Becker et al. 1996; Varman and Verma 1997]; here we consider the Multiversion B+-tree (MVBT) [Becker et al. 1996] which has optimal asymptotic behavior for range version retrievals.

The MVBT was proposed as a temporal access method. It keeps track of a set of temporal records, each having a key (in our case, the *start* of a record DNN) and a time interval. It efficiently supports the range version retrieval query by clustering objects both by their time intervals and by their keys. The MVBT is a graph structure that maintains the evolution of a B+-tree over time. It has many roots, each responsible for the subsequent part valid during a specific time interval. References to the different roots associated with the corresponding time intervals are kept in an additional data structure called *root\**. The MVBT partitions the key-time space into rectangles where each rectangle is associated with exactly one data page. A data record is stored in all the data pages whose key-time rectangle contains the data record's key and intersects the record's interval. The page rectangles are created recursively. As records are inserted into a certain page of a MVBT, the page may overflow. At that time, this page's currently alive data records are copied

to another page. The kind of copying is based on the number of alive records in the overflowed page. A *time-split* simply copies all alive records into a new page (Fig.5a). If many alive records exist, the time-split is followed by a *key-split* that distributes them into two new pages according to the median of their key attribute (Fig.5b).

Data records are inserted in the MVBT in increasing time order (i.e., *transaction-time* is assumed [Jensen and Snodgrass 1999]). When a data record is inserted at  $t$ , its deletion time is yet unknown and its interval is initiated to  $[t, now]$ ;  $now$  is a variable representing the ever increasing current time. For implementation purposes,  $now$  is stored as  $max.time$ . When later (if ever) this data record is deleted or updated, the  $end$  time in its interval is updated from  $now$  to the deletion time.

An important feature of the MVBT is that it guarantees a minimum key density for every page. In particular, for any time  $t$  in the page's rectangle, the page contains at least  $d$   $t$ -alive records, where  $d$  is linear to the page capacity. To achieve this, the MVBT uses yet another structural change: *merge*. If a *weak underflow* occurs after a deletion, i.e. the key density of the page where the deletion takes place drops below the threshold  $d$ , the alive records in the page and a sibling page are copied into a new page (Fig.5c). To avoid frequent merge/splits, the MVBT requires that when a new page is created, the number of records in it must be between a lower bound and a higher bound (*strong condition*). If the result page of a merge operation has too many records (more than the upper bound), a key split is performed immediately (Fig.5d).

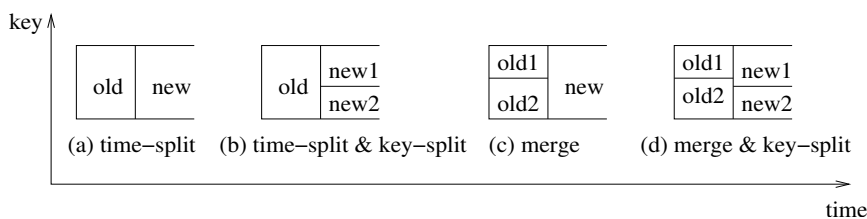


Fig. 5. Structural changes in a MVBT.

Below we compare the choice of MVBT against UBCC for the data storage organization. An advantage of the MVBT is that it offers efficient range 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 pre-allocate 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 (they contain the element's data as well).

## 5.2 Indexing Scheme Two: UBCC with Dense MVBT

In this scheme, the UBCC data organization is chosen, and an additional index is needed so as to support range version retrieval. The leaf pages of this additional index will store element SPaR 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, an MVBT secondary dense index can be used to implement the full-index. Various optimizations on this combination of UBCC and MVBT can be applied.

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 range version retrieval is accomplished by searching the MVBT using the given DNN range and version number(s). 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.

## 5.3 Indexing Scheme Three: UBCC with sparse MVRT

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 (1) store a set of records, each having a time interval (lifespan of the UBCC page) and a key range (the DNN range); and (2) support multiversion interval intersection queries: “given a key range  $r$  and a version  $t$ , find all UBCC page elements whose DNN ranges intersect  $r$  and whose lifespans contain  $t$ ”. The

previously discussed MVBT does not apply because it stores records with single keys, not with a key range. The *Multiversion R-tree (MVRT)* [Kumar et al. 1998; Tao and Papadias 2001] can be used instead.

A traditional (non-versioned) R-tree [Beckmann et al. 1990] is a spatial access method which stores a set of objects, each having a spatial region, and which supports efficiently the range query: “find elements which intersect a given spatial region”. In our case, if there is only one version, we can treat the UBCC page element as a 1-dimensional spatial object, where the spatial region of it corresponds to the DNN range. The range version retrieval query is reduced to the range query in the R-tree. To support multiple versions, we utilize the MVRT [Kumar et al. 1998; Tao and Papadias 2001] which 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.

Here we briefly review the MVRT. Consider a spatiotemporal evolution that starts at time  $t_1$  and assume that a 2-dimensional R-tree indexes the objects as they are at  $t_1$ . As the evolution advances, the 2D R-tree evolves too, by applying the evolution updates (object additions/deletions) as they occur. Storing this 2D R-tree evolution corresponds to making a 2D R-tree partially persistent. While conceptually the MVRT records the evolution of an ephemeral R-tree, it does not physically store snapshots of all the states in the ephemeral R-tree evolution. Instead, it records the evolution updates efficiently so that the storage remains linear, while still providing fast query time. The MVRT is actually a directed acyclic graph of nodes (a node corresponds to a disk page). Moreover, it has a number of root nodes, each of which is responsible for recording a consecutive part of the ephemeral R-tree evolution. Data records in the leaf nodes of a MVRT maintain the temporal evolution of the ephemeral R-tree data objects. Each data record is thus extended to include its lifespan, which is composed of *start* and *end* time. Similarly, index records in the directory nodes of a MVRT maintain the evolution of the corresponding index records of the ephemeral R-tree and are also augmented with a lifespan. An index or data record is alive for all time instants during its lifetime interval. With the exception of root nodes, a leaf or a directory node is called alive for all time instants that it contains at least  $D$  alive records ( $D = f \cdot B$ , where  $B$  is the maximum node capacity in number of records, and  $f < 1$  is a pre-determined constant). When the number of alive records falls below  $D$  (*weak-version underflow*), some action is needed. Here we have different choices. While [Kumar et al. 1998] proposed to split the node and copy its remaining alive records to another node, [Tao and Papadias 2001] proposed to re-insert the alive records into the current R-tree using the same idea as reinsertion in the R\*-tree. A range query regarding to region  $r$  at time  $t$  is performed as follows. First, the root node alive at  $t$  (there is exactly one such node in the MVRT) is located. Second, the objects intersecting  $r$  are found by searching this tree in a top-down fashion as in a regular R-tree. The lifespan of every record traversed should contain time  $t$ , and its MBR should intersect region  $r$ .

A difference of the scheme of using UBCC plus MVRT from the previous scheme of utilizing a dense MVBT 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 range version retrieval. In particular, it is desirable that the elements in the query range are reported in increasing DNN order, for the document tree structure can be reconstructed (via a stack mechanism) by scanning through the sorted list of objects. 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 copied 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.

## 5.4 Optimizations

In this section, we propose two optimizations to our indexing schemes for multiversion XML data.

**5.4.1 Range Deletion.** So far, we have made the assumption that each deletion is on a single element. To the best of our knowledge, this is the same assumption made in all the existing temporal access methods like the MVBT or the MVRT. However, this is not efficient to handle XML documents. If the user wants to delete a whole chapter, the system will have to find all objects (sections, subsections, titles, figures, and so on) in the chapter and then delete every object separately. Mapped to the temporal database environment, the operation can be expressed formally as:

*Definition 5.1.* Given a set of temporal objects, the *range deletion* operation with regards to range  $r$  logically deletes all objects alive in the current version whose keys are in  $r$ .

Existing index structures should be extended to more efficiently support the range-deletion operation. The idea is to allow deletion to stop at higher levels of the tree. Take the MVRT as an example, which is used in our scheme 3 to index the DNN ranges and lifespans of all UBCC pages, and assume that we want to delete all objects in the current version whose DNNs are within a given DNN range  $r$  (e.g. corresponding to chapter 3). We search the current version of the MVRT. Whenever we find an index entry whose DNN range is contained in  $r$ , we mark the index entry as being deleted, without examining the sub-tree. Similarly, if we find a leaf entry in the MVRT whose DNN range is contained in  $r$ , we mark the entry as being deleted at the current version, without fetching the corresponding UBCC page into memory and logically delete all alive objects in the page.

To extend the MVBT (used in scheme 1 & 2) to handle range deletion, we need to extend the merge and merge-split structural changes to involve more than two input pages. Recall that in Fig. 5, if a page in the MVBT experience weak-version underflow, i.e. it does not have enough records alive in the current version, it is merged with a sibling page. Now, a range deletion may results in weak-version underflow of multiple pages. They should be merged into one. Of course, if the newly generated page experiences strong-version overflow, it is split into two.

**5.4.2 Incremental Updates.** So far, we have assumed that an object update is handled by introducing a new copy of the object. While simple, this approach leads to high storage complexity, which adversely affects the query performance. This is true especially for small updates. In a practical environment, from one version to the next, we expect to see many incremental updates. That is, the content of an object (e.g. the text part of an sub-section) changed but only a little. If for example, only 1% of the object content has changed but we store a completely new copy of the object, space is wasted.

In this case, we should store the two copies of the object together, where for the new copy we only store the difference from the original full copy. If the new copy is needed upon query, we construct it from the full copy and the difference. This idea extends to multiple updates straightforwardly. We keep one copy and a list of changes. If any non-stored copy is needed, we construct it by reversely applying the changes starting from the saved copy. To compute the difference between two objects and to reversely apply the changes to construct a needed version, standard utilities like *xdelta*, *patch* and *diff* may be used.

When we copy an alive object (which may be associated with many changes) to a new page, we just copy the object itself and not the changes. This is important since otherwise the number of changes associated with an alive object keeps increasing, which leads to large storage size and longer reconstruction time.

## 6. PERFORMANCE EVALUATION

To evaluate the three choices for data organization and full-index implementation we present results comparing their query time performance for two variations of the range version retrieval query, namely single-version and multiple-version retrievals. All schemes are also compared in space (index size) and update (check-in time) performance.

## 6.1 Experimental Setup

Scheme 1 uses a single dense primary MVBT, scheme 2 uses UBCC plus a dense secondary MVBT (section 5.2), while scheme 3 uses UBCC plus a sparse MVRT index (section 5.3). The usefulness in both UBCC based schemes was set to 0.5.

The schemes were implemented in C++ using GNU compilers under Cygwin. The programs were run on a Dell Optiplex GX260 machine running Windows XP Professional, with 2.66GHz Pentium(R) IV CPU with 1.0GB memory and 80GB 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 5ms. We used a 16KB page size. For all the three schemes we used the LRU buffering where the buffer size was 100 pages.

Due to the lack of real datasets for multi-versioned XML documents, we use the following semi-realistic dataset: the initial version of our document is the Hamlet play from the Shakespeare dataset [Bosak 1999]. It contains 6636 elements, the longest of which has 659 characters. The depth of the tree is 6. 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 5.2). 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.)

## 6.2 Check-in Time and Index Size

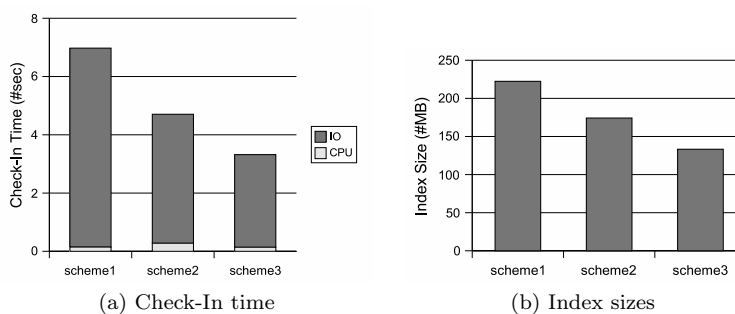


Fig. 6. Comparing the check-in time and the index sizes.

Fig.6a compares the check-in time per version of the three schemes, while Fig.6b compares the index sizes. The version check-in time measures the total time to finish all the updates within a given version. It is averaged across the 1000 versions. Scheme 3 has the fastest check-in time and uses the least index size. Similarly,

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 copies 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.

### 6.3 Single Version Retrieval Queries

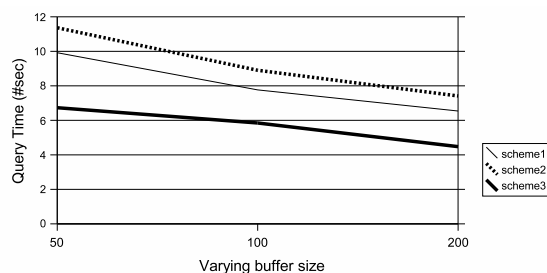


Fig. 7. Performance for the full-document single-version retrieval query.

To evaluate the query performance of the three schemes, we first compare their behavior for the full-document single-version retrieval query. We measured the average execution time of 10 randomly generated versions, varying the buffer size in number of disk pages. As shown in Fig.7, Scheme 3 is the best. For a full-document retrieval, all the UBCC pages that are useful at the query version need to be examined. In this case, Scheme 3 spends the least amount of time in finding these UBCC pages (as it uses a smaller sparse index). Between Scheme 1 and Scheme 2, we observe that Scheme 2 has worse performance, even though it has better check-in time and smaller index. This is because in Scheme 2, the actual objects and the references to these objects are maintained in two different structures and thus they have different clustering. To perform a query, we first need to find the references to the objects from the MVBT. A separate stage is needed to locate the referenced actual objects from the UBCC pages. Note that multiple references located from the MVBT may correspond to the same UBCC page. In order to avoid accessing one UBCC page multiple times, we sort the references based on UBCC page number before accessing the UBCC file.

	CPU	I/O	Total
Scheme 1	1	10	11
Scheme 2	2	16	18
Scheme 3	5	212	217

Table I. Query time (#ms) comparison for the single-element single-version retrieval query.

We also examined the other extreme case, i.e., single-element retrieval in a single version. Again, we measure the average execution time of 10 randomly generated queries with a very small DNN range. The performance comparison (in msec) is shown in Table I. In this case, Scheme 3 (the previous winner) is not as efficient as the other two schemes. The reason is that scheme 3 may check some UBCC pages which do not contain any qualifying object. In more detail, it is likely that the query range intersects the DNN range of a UBCC page, but the page does not contain any object whose DNN is in the query range. But since the sparse index MVRT only stores the information for UBCC pages, not information of individual objects in the pages, Scheme 3 needs to access such pages. This is not the case for schemes 1 and 2. Scheme 1 is the best for single-element retrieval since it directly finds the qualifying object.

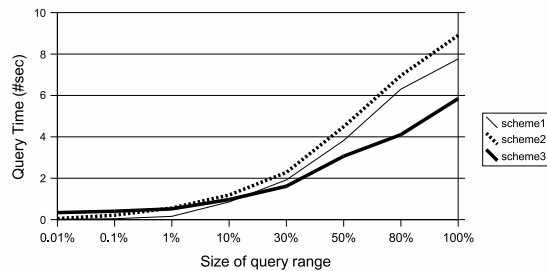


Fig. 8. Partial-document single-version retrieval query performance with varying length of DNN range.

Fig.8 compares the three schemes on the partial-document single-version retrieval query while varying the length of query DNN range. For each length of DNN range, we measure the average execution time of 10 randomly generated queries. 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. For small query DNN ranges, all schemes have relatively small and comparable response time, with Scheme 1 being the faster.

#### 6.4 Multiple Version Retrieval Queries

We next experimented with multiple version retrieval queries, that is, queries requesting multiple subsequent versions; this is typically the case for version-interval and version aggregate queries. Again we differentiated between full-document, partial-document and single-element queries.

For the full-document retrieval we vary the length of the version interval (i.e., how many consecutive versions are requested). An example is: “find the document between versions 10 and 20”. The result is shown in Fig.9. Scheme 3 consistently has better performance than the other competitors.

Fig 10 depicts the single-element multiple versions query performance results, while varying the length of the version interval. As in the single version case (Table

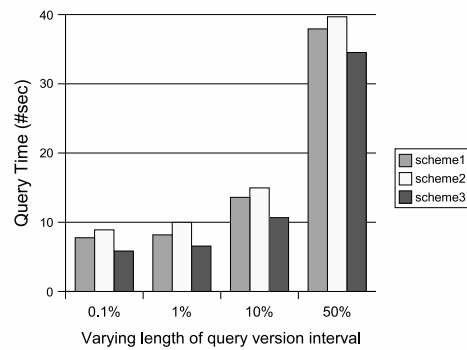


Fig. 9. Full-document, multiple-version query performance with varying version interval length.

I), scheme 1 is the best, while scheme 3 is worse. Again, the reason is that scheme 3 may check various UBCC pages which do not contain any qualifying object.

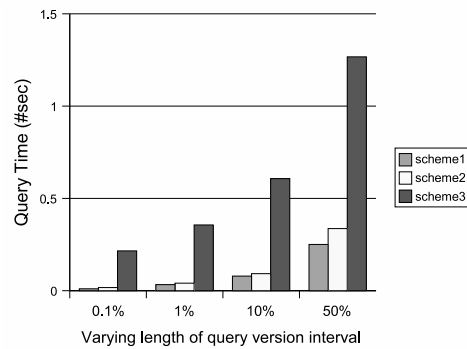


Fig. 10. Single-element, multiple-version query performance with varying version interval length.

Finally, Fig. 11 presents the partial-document multiple-versions query while varying the length of DNN range. The number of subsequent versions requested was fixed to be 10% of the total number of versions. As with Fig.8, for small DNN range (e.g. corresponding to few elements) all schemes are competitive, however as the query DNN range increases, scheme 3 is the most efficient.

In conclusion, considering check-in time, storage, and query time, the UBCC plus sparse MVRT (scheme 3) shows the most robust performance.

## 7. CONCLUSIONS

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, there is the efficient execution of complex queries, such as the path expression queries which are now part of the XML query languages proposed as standards

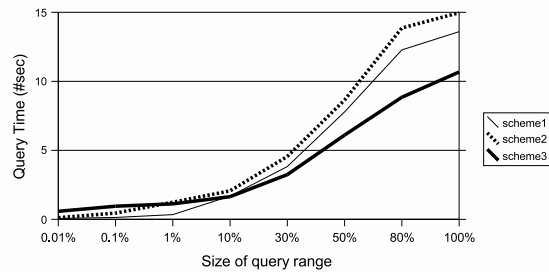


Fig. 11. Version-interval query performance, fixing the version-interval to be 10% of the total number of versions and varying the length of DNN range.

[WWW Consortium 1999; 2001]. 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 range version retrieval queries; then, we evaluated alternative indexing and clustering schemes for the efficient execution of range 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).

#### APPENDIX: The Usefulness-Based Clustering Control

The usefulness-based clustering is similar to a technique used in transaction-time databases [Lomet and Salzberg 1989; Tsostras and Kangelaris 1995; Becker et al. 1996] to cluster temporal data and is outlined below. 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 [Chien et al. 2000] uses the edit script to determine the useful pages for each version, while [Chien et al. 2001] facilitates the object references. Full version retrieval under the SPaR scheme is discussed in section 3.3.

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.

### Acknowledgments

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

### REFERENCES

- ABITEBOUL, S., KAPLAN, H., AND MILO, T. 2001. Compact Labeling Schemes for Ancestor Queries. In *Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.
- ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., AND WIENER, J. L. 1997. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries* 1, 1, 68–88.
- AL-KHALIFA, S., JAGADISH, H. V., KOUFAS, N., PATEL, J. M., SRIVASTAVA, D., AND WU, Y. 2002. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of International Conference on Data Engineering (ICDE)*. 141–154.
- BECKER, B., GSCHWIND, S., OHLER, T., SEEGER, B., AND WIDMAYER, P. 1996. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal* 5, 4, 264–275.
- BECKMANN, N., KRIEGEL, H. P., SCHNEIDER, R., AND SEEGER, B. 1990. The R\*-tree: An Efficient ACM Journal Name, Vol. V, No. N, M 2004.

- and Robust Access Method for Points and Rectangles. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*. 322–331.
- BEECH, D. AND MAHBOD, B. 1988. Generalized Version Control in an Object-Oriented Database. In *Proceedings of International Conference on Data Engineering (ICDE)*.
- BOSAK, J. 1999. The Plays of Shakespeare in XML. <http://www.oasis-open.org/cover/bosakShakespeare200.html>.
- BRUNO, N., KOUDAS, N., AND SRIVASTAVA, D. 2002. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*. 310–321.
- BUNEMAN, P., KHANNA, S., TAJIMA, K., AND TAN, W. 2002. Archiving Scientific Data. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*.
- CERI, S., COMAI, S., E. DAMIANI, P. FRATERNALI, S. P., AND TANCA, L. 1999. XML-GL: A Graphical Language for Querying and Restructuring XML. In *Proceedings of WWW Conference*. 1171–1187.
- CHAMBERLIN, D., ROBIE, J., AND FLORESCU, D. 2000. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of International Workshop on the Web and Databases (WebDB)*. 1–25.
- CHAWATHE, S. AND GARCIA-MOLINA, H. 1998. Representing and Querying changes in semistructured data. In *Proceedings of International Conference on Data Engineering (ICDE)*.
- CHAWATHE, S., RAJARAMAN, A., GARCIA-MOLINA, H., AND WIDOM, J. 1996. Change Detection in Hierarchically Structured Information. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*. 493–504.
- CHIEN, S.-Y., TSOTRAS, V. J., AND ZANIOLO, C. 2000. Version Management of XML Documents. In *Proceedings of International Workshop on the Web and Databases (WebDB)*. 184–200.
- CHIEN, S.-Y., TSOTRAS, V. J., AND ZANIOLO, C. 2001. Efficient Management of Multiversion Documents by Object Referencing. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 291–300.
- CHIEN, S.-Y., TSOTRAS, V. J., AND ZANIOLO, C. 2002. Efficient schemes for managing multiversion XML documents. *VLDB Journal* 4, 332–353.
- CHIEN, S.-Y., TSOTRAS, V. J., ZANIOLO, C., AND ZHANG, D. '01b. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In *Proceedings of International Conference on Web Information Systems Engineering (WISE)*. 232–244.
- CHIEN, S.-Y., TSOTRAS, V. J., ZANIOLO, C., AND ZHANG, D. '02b. Efficient Complex Query support for Multiversion XML documents. In *Proceedings of International Conference on Extending Database Technology (EDBT)*. 161–178.
- CHIEN, S.-Y., VAGENA, Z., ZHANG, D., TSOTRAS, V. J., AND ZANIOLO, C. '02c. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 263–274.
- COHEN, E., KAPLAN, H., AND MILO, T. 2002. Labeling Dynamic XML Trees. In *ACM International Symposium on Principles of Database Systems (PODS)*.
- DEUTSCH, A., FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. 1999. A Query Language for XML. In *Proceedings of WWW Conference*. 1155–1169.
- DIAO, Y., FICHER, P., FRANKLIN, M., AND TO, R. 2002. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of International Conference on Data Engineering (ICDE)*.
- FERNANDEZ, M. AND SUCIU, D. 1998. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of International Conference on Data Engineering (ICDE)*. 14–23.
- FIEBIG, T. AND MOERKOTTE, G. 2000. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proceedings of International Workshop on the Web and Databases (WebDB)*.
- GOTTLOB, G., KOCH, C., AND PICHLER, R. 2002. Efficient Algorithms for Processing XPath Queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*.
- GOTTLOB, G., KOCH, C., AND PICHLER, R. 2003a. XPath Processing in a Nutshell. *SIGMOD Record* 32, 1, 12–19.

- GOTTLob, G., KOCH, C., AND PICHler, R. 2003b. XPath Query Evaluation: Improving Time and Space Efficiency. In *Proceedings of International Conference on Data Engineering (ICDE)*.
- HALVERSON, A., BURGER, J., GALANIS, L., KINI, A., KRISHNAMURTHY, R., RAO, A., TIAN, F., VIGLAS, S., WANG, Y., NAUGHTON, J., AND DEWITT, D. 2003. Mixed Mode XML Query Processing. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*.
- JENSEN, C. AND SNODGRASS, R. 1999. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 11, 1, 36–44.
- JIANG, H., LU, H., WANG, W., AND OOI, B. C. 2003. XR-Tree: Indexing XML Data for Efficient Structural Join. In *Proceedings of International Conference on Data Engineering (ICDE)*.
- JIANG, L., SALZBERG, B., LOMET, D., AND BARRENA, M. 2000. The BT-tree: A Branched and Temporal Access Method. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 451–460.
- KAPLAN, H., MILO, T., AND SHABO, R. 2002. A comparison of Labeling Schemes for Ancestor Queries. In *Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*.
- KATZ, R. H. AND CHANGE, E. 1987. Managing Change in a Computer-Aided Design Databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 455–462.
- KUMAR, A., TSOTRAS, V. J., AND FALOUTSOS, C. 1998. Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 10, 1, 1–20.
- LANDAU, G. M., SCHMIDT, J. P., AND TSOTRAS, V. J. 1995. Historical Queries Along Multiple Lines of Time Evolution. *VLDB Journal* 4, 4.
- LANKA, S. AND MAYS, E. 1991. Fully Persistent B+ Trees. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*. 426–435.
- LEBLANG, D. 1994. The CM Challenge: Configuration Management that Works. In *Configuration Management*, W. F. Tichy, Ed. Wiley Co., 1–38.
- LEVY, A., FLORESCU, D., SUCIU, D., KANG, J., AND FERNANDEZ, M. 1997. STRUDEL - a Web-site management system. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*.
- LI, Q. AND MOON, B. 2001. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 361–370.
- LOMET, D. AND SALZBERG, B. 1989. Access Methods for Multiversion Data. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*. 315–324.
- MARIAN, A., ABITEBOUL, S., COBENA, G., AND MIGNET, L. 2001. Change-Centric Management of Versions in An XML Warehouse. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 581–590.
- McHUGH, J. AND WIDOM, J. 1999. Query Optimization for XML. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 315–326.
- ÖZSOYOGLU, G. AND SNODGRASS, R. 1995. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 7, 4 (Aug.), 513–532.
- RAO, P. AND MOON, B. 2004. PRiX: Indexing and Querying XML Using Prifer Sequences. In *Proceedings of International Conference on Data Engineering (ICDE)*.
- ROCHKIND, M. J. 1975. The Source Code Control System. *IEEE Transactions on Software Engineering SE-1*, 4 (Dec.), 364–370.
- SALZBERG, B. AND TSOTRAS, V. J. 1999. Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys* 31, 2, 158–221.
- SHANMUGASUNDARAM, J., TUFTE, K., HE, G., ZHANG, C., DEWITT, D. J., AND NAUGHTON, J. F. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 302–314.
- TAO, Y. AND PADIADIS, D. 2001. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 431–440.
- TATARINOV, I., VIGLAS, S. D., BEYER, K., SHANMUGASUNDARAM, J., SHEKITA, E., AND C. ZHANG. 2002. Storing and Querying Ordered XML using a Relational Database System. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*.

- TIAN, F., DEWITT, D. J., CHEN, J., AND ZHANG, C. 2002. The Design and Performance Evaluation of Various XML Storage Strategies. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*. 5–10.
- TICHY, W. F. 1985. RCS—A System for Version Control. *Software—Practice & Experience* 15, 7 (July), 637–654.
- TSOTRAS, V. J. AND KANGELARIS, N. 1995. The Snapshot Index: An I/O-Optimal Access Method for Timeslice Queries. *Journal of Information Systems* 20, 3, 237–260.
- VAGENA, Z., MORO, M., AND TSOTRAS, V. J. 2004. Supporting Branched Versions on XML Documents. In *Proceedings of International Workshop on Research Issues on Data Engineering (RIDE)*.
- VAGENA, Z. AND TSOTRAS, V. J. 2003. Path-expression Queries over Multiversion XML Documents. In *Proceedings of International Workshop on the Web and Databases (WebDB)*.
- VARMAN, P. AND VERMA, R. 1997. An Efficient Multiversion Access Structure. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 9, 3, 391–409.
- WANG, H., PARK, S., FAN, W., AND YU, P. 2003. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data (SIGMOD)*.
- Webdav 2001. webdav, WWW Distributed Authoring and Versioning, last modified: Jul 31, 2001. <http://www.ietf.org/html.charters/webdav-charter.html>.
- WEI, W., HAIFENG, J., LU, H., AND YU, J.-X. 2003. PBiTree Coding and Efficient Processing of Containment Join. In *Proceedings of International Conference on Data Engineering (ICDE)*.
- WONG, R. AND LAM, N. 2002. Managing and Querying Multi-Version XML data with Update Logging. In *Proc. of DocEng*.
- WWW Consortium 1999. XML Path Language (XPath), version 1.0. <http://www.w3.org/TR/xpath.html>.
- WWW Consortium 2001. XQuery 1.0: An XML Query Language. *W3C Working Draft (work in progress)*, <http://www.w3.org/TR/xquery>.
- ZHANG, D., MARKOWETZ, A., TSOTRAS, V. J., GUNOPULOS, D., AND SEEGER, B. 2001. Efficient Computation of Temporal Aggregates with Range Predicates. In *ACM International Symposium on Principles of Database Systems (PODS)*.

Received Month Year; revised Month Year; accepted Month Year