Supporting Branched Versions on XML documents

Zografoula Vagena, Mirella M. Moro, Vassilis J. Tsotras
Computer Science and Engineering, University of California, Riverside
Riverside, CA 92521
{foula, mirella, tsotras}@cs.ucr.edu

Abstract. Many e-commerce and e-government applications are collaborative in nature (e.g. negotiation and e-catalog management). In collaborative environments, users typically define new document versions from any past version, which creates the need for supporting multiversion XML documents, particularly *branched versioning*. In this paper, we address the problem of evaluating path expression queries over XML documents with branched versions. We extend path joins to work in a branched version environment and to allow queries on multiple versions. We propose a storage scheme that efficiently maintains all branched document versions and describe changes required on Pathstack, an optimal pattern matching algorithm. Finally, we investigate the effectiveness of our techniques through experimental evaluation.

1 Introduction

With the growth of the Web and e-commerce applications, XML has become the *de facto* standard for authoring, storing, exchanging and presenting documents [32]. Gradually, a number of systems are emerging, providing facilities to store XML data, either natively, or in one of the existing relational or object-oriented repositories. Like any other data source, a document or an XML database is often subject to updates [26]. In the "ephemeral" (i.e., non-versioned) case, an update creates a new document to be viewed and queried, while the document before the update is not retained. However, in several environments (typically collaborative scenarios, e.g. document exchange in business negotiation, e-catalogs management, configuration management, CAD design, etc) users are interested not only in the current state of the document but also previous ones. For example, an old version may prove to have better features than the latest one. Moreover, users may also be interested in querying the document evolution (i.e., *deltas*) [21, 7, 6, 5]. Tools and techniques that facilitate such *version management* are thus important.

In the simplest version management scheme, a new version is created out of the currently latest version; this results to a single line of consecutive versions (linear versioning). In practice, however, most design applications follow the *branched* versioning scheme. Here a new version can be derived from any previous one, creating a "tree of versions". Consider as an example a business negotiation process. Negotiation may be defined as a sequence of message exchanges in order to agree on a business contract [23]. Electronic negotiations involve the exchange of structured electronic messages and documents, defined in XML for example. There is a document as the first contract offer (version) between the business partners. During the negotiation, new documents (i.e., new contract versions) are created through message exchanges between the negotiators. Hence, as the negotiation process advances, information in the contract is changed leading to a new version of the contract, as shown in Figure 1. It is possible to discard a current version of the contract and reuse an old version as well [25]. Furthermore, one of the parts may create (parallel) children versions (e.g. with different prices and deadlines) from the same document as options to be evaluated by the other part. In Figure 1, versions 3, 4 and 5 are children of version 2.

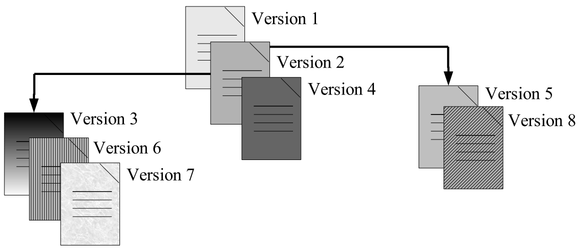


Figure 1: Contract versions in negotiation process example.

Another example appears with e-catalogs. Electronic catalogs have become the entry point for both consumer-oriented and business-to-business oriented e-commerce[2]. Catalog data has usually a master data character (e.g. technical product features), while specific data changes over time (e.g. prices) or can be determined only time-dependently [18]. For instance, a company sets different prices for specified periods due to promotional seasons, product demand, and large orders may benefit from special discount as well. The same product may have different versions according to pre-defined situations. Furthermore, one user may need only the prices for some products. Existing systems return the whole document, and the user has to go through it looking for the desired information.

In a large warehouse where documents are updated frequently (e.g. e-catalogs), storage requirements can grow fast. In one extreme, each new version is stored as a whole ("snapshot" approach). This results in fast query performance but consumes high storage (quadratic increase). In the other extreme, the very first document version is stored intact, with all later version updates appended in a log ("log" approach). While having minimal space requirements, the "log" approach has high query time since answering a query about version v involves reconstructing multiple previous versions that led to version v. The effect on performance is even worse if the query is concerned only with portions of version v. Instead we aim at a versioning solution that combines the advantages from each extreme: the fast query processing of the "snapshot" with the linear space of the "log" approach.

Previous work on XML versioned documents has concentrated in efficient reconstruction of a whole, single version, and at best has considered linear versioning only. In practice however, the user is interested in querying parts of an XML document. This is accomplished by supporting "path expression" queries, which involve patterns of selection predicates on multiple document elements with specified structural relationships. For non-versioned documents, path expression queries have been evaluated as path joins.

An optimal pattern matching algorithm (Pathstack) has been introduced in [4].

This paper addresses path expression queries on XML documents with branched versions. We first propose a structure to efficiently store document versions so that path queries on any version(s) can be efficiently evaluated. We then provide a non-trivial extension to the Pathstack algorithm in order to address path queries in the multiversion environment (we call the new algorithm the *Version Range Pathstack*, VRP). The original Pathstack algorithm would require a stored snapshot of each version, leading to prohibitively large space. Also, it does not deal with branched versions.

When multiple queries access the version warehouse, it would be beneficial to take advantage of the common parts of the queried versions such that these parts are accessed only once for all versions. We thus examine two forms of path expression queries that may include *many* versions. That is, the query is evaluated either: (i) among a group of versions belonging to the same path in the version tree (e.g. evaluate query `a//b//c` for versions 1, 2, and 4 in Figure 1), or, (ii) among a group of versions that share the same parent, along with the parent version (evaluate `a//b//c` for versions 2, 3, and 5). Combinations of these primitive queries are possible enabling the user to easily navigate the version tree.

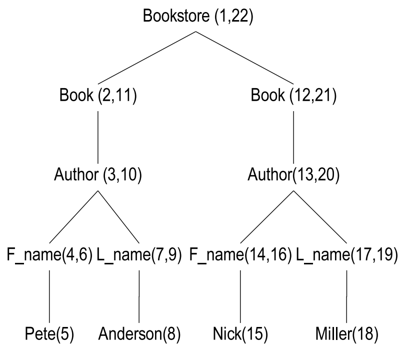
The rest of the paper is organized as follows. Section 2 provides related work from temporal data management, version management and structural query processing. Section 3 presents the proposed techniques, while section 4 contains the experimental evaluation. Finally, section 5 concludes the paper.

2 Background and Related Work

XML model. An XML warehouse is modeled as a forest of rooted, ordered, labeled trees, where each tree represents a document. Each node in the tree corresponds to an element or a value, and the edges represent immediate element-sub element or element-value relationships. Figure 2a shows an XML document for the e-catalog of a bookstore, its two books and their authors. Figure 2b presents the tree structure for this document (including a range-based numbering scheme to be discussed next).

```
<Bookstore>
  <Book>
    <Author>
      <F_name>Pete</F_name>
      <L_name>Anderson</L_name>
    </Author>
  </Book>
  <Book>
    <Author>
      <F_name>Nick</F_name>
      <L_name>Miller</L_name>
    </Author>
  </Book>
</Bookstore>
```

(a)



(b)

Figure 2: XML model.

Structural Joins. XML queries (e.g. [31]) typically specify patterns of selection predicates on multiple elements that have some specified tree structure relationships. Path expressions are the building components for such queries and are thus at the core of XML query processing. A path expression defines a series of XML tree node labels, every two of which are related with an ancestor-descendant or parent-child relationship. Various works have recently addressed efficient implementation of such queries on non-versioned documents when these documents either reside in main memory [13] or on the disk [1, 4, 9, 14, 19, 28]. Our work relates to the second group, as it deals with large warehouses that are disk resident.

A usual assumption for the disk-based approaches is that a numbering scheme [10, 1, 19, 12] is embedded on the document tree nodes in order to capture their structural relationships. One commonly used numbering scheme is *range-based*; each node in the document tree is assigned a range: (*left*,*right*). Given a node pair (a, d), node d is a *descendant* of node a (which is then an *ancestor* of d), if and only if, $a.left < d.left < d.right < a.right$. For example, node L_name with range (17,19) is a descendant of node Book(12,21) but is not descendant of node Author(3,10) (Figure 2b). Any two node ranges can either be disjoint or one fully covers the other. Hence the $left$ (and $right$) numbers are unique for each node, and can be used to identify the position of the node in the tree. Since leaf (value) nodes in the XML tree do not have children, they are assigned a single number instead of a range, e.g. Miller(18). To capture parent-child relationships the tree level is also used (not shown in Figure 2b).

Using range-based numbering, path-expression queries reduce to path joins [1, 4]. Holistic Joins ([4]) represent the state of the art in optimally processing path joins. Durable (i.e., not affected by document changes), range based, numbering schemes have also been used [19, 8]. Here, unused numbering ranges are left open for use with future element additions. Nevertheless, renumbering may be needed at the worst case that a preassigned range runs out of numbers. Updatable, prefix-based numbering schemes have been also proposed [11]; 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. Previous work on version management of semi-structured documents [6] and XML in particular [7, 21, 8, 5, 29, 27], has considered various aspects of the storage and manipulation of multiversion documents. [7] proposes techniques that represent the document evolution as edit scripts or object references with the focus on fast reconstruction of any particular document version. [21] considers queries on the XML document's evolution changes. [5] concentrates on the compact storage of a XML archival system and its version querying assumes the existence of key identifiers. [8] addresses "simple" path expression queries (i.e., the special case where an element cannot appear in the subtree of an element with the same type) as combinations of partial version retrieval queries. However, none of the above approaches is optimized for general path expression queries in a multiversion XML environment. Recently, we have proposed algorithms for general path-expression queries for the linear versioning case [27]. Here we extend that work in two ways: (i) we provide solutions for the more complex branched versioning problem, and, (ii) we address queries that may involve more than one version.

Linear versioning is similar to the notion of "transaction time" in temporal databases [22]. A number of data structures has been proposed to index transaction-time databases [24]. Such linear-versioning methods (like the Multiversion B-tree [3], or the Time-Split B-tree [20]) are typically directed acyclic graphs of index and data (leaf) pages. All data pages are at the same distance from their root. Records have a key and a version interval (v_i, v_j), where v_i is the version this record was created and v_j the version it was deleted (if ever) – for simplicity in the rest we consider node additions and deletions only; node updates can be presented with a deletion followed by an addition.

Pages containing such records are assigned a key range K, a version range R, and store records whose key and start version are in (K, R). A page may store records valid for various consecutive versions. However, if a page contains at least one record from a given version, it is guaranteed to have enough number of records valid for that version. The records valid for a given version are clustered in few pages.

When a page p overflows, a *version split* is performed: records in p that are valid for the current version are copied to a new page q. Subsequently, to prevent immediate future page overflows for page q, if the number of valid records in q is above an *upper* threshold ut, a key split is performed. That is, the records in q are distributed among q and another new page. Such split may be propagated to index pages.

A page p can become *invalid* (or underutilized) for a given version v if the number of records in p valid for version v falls below a *lower* threshold lt ($lt < ut$). This may happen as records in this page are deleted/updated. Page p is then consolidated with a sibling page, and records from both pages that are valid for version v are copied to a new page. A key split may further be performed.

Fewer works have addressed issues related to version branching [17, 16, 15]. Among these, [17] proposes the fully-persistent B-trees, an indexing method for branched versioned data, while [16] discusses querying techniques in this environment. Recently, [15] proposes access methods to efficiently store branched and temporal data, by providing a reasonable trade off between space consumption and performance of queries over any single version. It uses similar clustering techniques adjusted for branched versioning. Instead of the BT-tree [15], our work proposes a branched and temporal element list (we call it the BT-element list), which is further optimized for the case where multiple versions need to be accessed at once.

3 Path Expression Query Processing

This section presents the data structures and techniques for path queries over branched documents versions.

3.1 Data Structures

Version Tree. We maintain a tree structure that represents the graph of versions. Each node of the tree corresponds to one version. Typically this tree is small and can be kept in main memory. Figure 3a presents different versions of an XML document, while figure 3b depicts the corresponding version tree.

If the number of versions grows such that the tree cannot be held in main memory, the tree can be migrated to the disk by declaring a *version tree node* relation. Each row in the relation keeps information associated with a single tree node. Moreover, to speed up query time, given two versions, we need to quickly verify ancestor-descendant relationships (see discussion in section 3.2). Such checks can be performed with a numbering scheme on the version tree. Note that this is a different numbering from the numbering of the XML document. As a result, a row in the relation is of the form *<versionID, numbering scheme for version tree>*. Such a numbering scheme is also easily implemented when the version tree resides in main memory and considerably saves computation time.

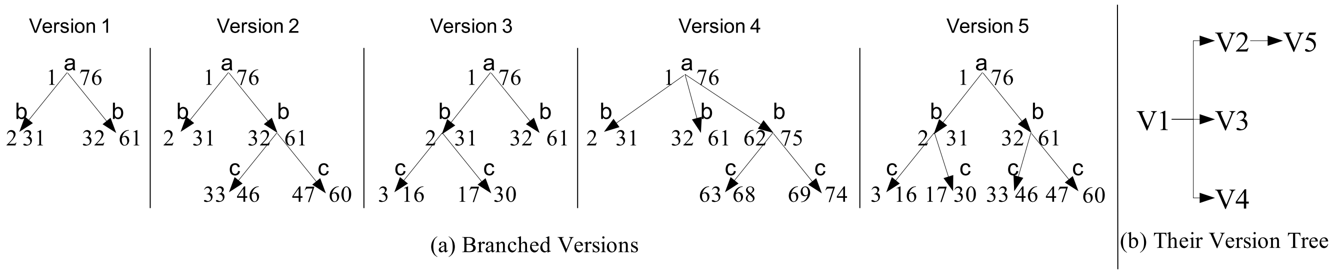


Figure 3: Document versions and the corresponding Version Tree.

Data Records. Each element node in a document tree is stored as a record that contains: (i)the node content, (ii) the *left* and *right* numbers, and, (iii) the version identifier (this is the version where this record was inserted into the document tree). In a linear versioning system, each node's record is assigned a single version "lifetime" interval (v_i, v_j), where v_i is the insertion version and v_j the deletion version (if ever) for this node. However, in the branched versioning we consider, a node may follow various parallel version lifetimes, as it may be modified/deleted by many subsequent versions. A naive way to keep the version lifetimes would be to maintain an entry in the node record for each version where the record is deleted. However, this is not practical as the number of such versions is unknown and could be unbounded. We adopt a simpler scheme, where we insert a special "tombstone" record at each version where this node is deleted. A tombstone record stores the node's (uniquely identifier) *left* number, the deletion version and a special value denoting that it is a tombstone record. An interesting distinction with regular records is that tombstone records are not replicated during version split operations.

The BT-ElementList. There are various ways to cluster the versions of a document, depending on which query the user needs to optimize. For the path expression queries, the Pathstack algorithm, assumes that document nodes are organized in element lists. That is, the document is vertically partitioned by element type. A separate list exists containing all nodes of the same type, and each such list is sorted by the *left* number of the nodes it stores. The problem remaining is how to efficiently maintain and access the various states that each element list acquires as the document evolves through versions. Given a list L, we want to be able to access the ordered contents of L as of a given version v. To do so, we propose a data structure called the BT-ElementList. (Note that this is a structure simpler than the BT-tree [15]). Moreover, the reconstruction of the whole document at a particular version is still possible, by simply performing a merge scan over the element lists as of that version.

Optimizations. We offer two optimizations specific for the path expression queries we address. First, as such queries involve accessing multiple versions of an element list, we redefine when it is justified to access a page. That is, a page is accessed as long as it contains enough valid records for the *union* of the queried versions. This affects the version clustering mechanism and results in less copying. A page is copied less often as it may still be considered valid, if it has enough valid records for a number of versions. Second, we introduce an additional version split algorithm that effectively reduces the replication between ancestor-descendant versions in the version-tree.

Page 0	Page 1	Page 2	Page 3	Page 4	Page 5	Page 6
1,1,...	1,2,...	32,2,...	1,3,...	3,3,...	1,4, ...	32,4,...
2,1,...	2,2,...	33,2,...	2,3,...	17,3,...	2,4,...	62,4,...
32,1...		47,2,...		32,3...		63,4,...
						69,4,...

Lower Threshold = 50%
(a) Version split in common records

Page 0	Page 1	Page 2	Page 3
1,1,...	32,2,...	3,3,...	32,4,...
2,1,...	33,2,...	17,3,...	62,4,...
32,1...	47,2,...	32,3,...	63,4,...
			69,4,...

Lower Threshold = 50%
(b) Alternate version split

Figure 4: Effect of version split in common records and application of alternate version split

The first optimization improves the space required for the version archive by carefully consolidating underutilized pages. Let i be the number of valid records for version v in page p. When i falls below the *lower* threshold (lt), we also check the number of valid records in p for the parent (v_a) and child versions (v_c) of v. Let j be the minimum among the number of valid records from v_a and the possibly many v_c's. Page p is considered invalid if $i + j < lt$. The intuition for this heuristic is that, in general, whenever we access a page, we also access either its parent or a child. Therefore, access to this page is still justified, since we collect a large number of valid records.

The advantage of the second optimization is explained through figures 4a and 4b. Figure 4a depicts the pages of the archive after creating the fourth version in the example of Figure 3. For simplicity, each record shows only its *left* number and version identifier; in this example $ul = 90\%$ and $lt = 50\%$ of the page size. Observe that copies of records with *left* number 1 and 2 appear in pages 0, 1, 3 and 5. Version 1 has been stored in page 0. Version 2 adds elements 33 and 47, which cause page 0 to overflow. As a result, a version split occurs and since all records are valid, a key split follows. This creates pages 1 and 2. Version 3 derives from version 1, so it attempts to add records 3 and 17 in page 0; a version and key split create pages 3 and 4. Similarly, version 4 results to pages 5 and 6. Clearly, all version splits were followed by a key splits, creating 2 new pages at each split. To limit this record duplication, we introduce the constraint that when a page overflows if we have more records than *twice* the lower threshold lt, we create a single new page, in which we put only as many records necessary to keep it above lt. Note that the original page remains above threshold lt as well. Figure 4b illustrates the same archive after version 4, with the modified version split procedure.

To access each element list in order, every page keeps track of its next page per version, by using a fixed space area within each page. Whenever this space is consumed, a new page is allocated and the records that belong to half of the versions are copied there. This is an extension of the persistent lists of [7]. Due to lack of space, details are omitted for the full version of this paper.

3.2 Path Queries over Branched Document Versions

We start with a short description of the **PathStack** algorithm [4]. Given a path query, say `a/b/c`, PathStack associates with every query element type (in this case `a,b,c`) a *stack*. The algorithm's input are the ordered element lists of the query elements. As the algorithm proceeds, elements are taken from their corresponding element lists and are pushed in (and subsequently popped from) their stacks. Moreover, each element in a stack points to an element in the stack of the parent type, as the latter is defined by the query (e.g. elements in the stack for type `c` point to elements in the stack for type `b`). There are three computation invariants in the algorithm: (a) Moving from the bottom to the top of a given stack, the elements correspond to document nodes that belong to the same path in the XML tree. (b) Each element in a given stack corresponds to an XML tree node that is an ancestor of all elements above it in the stack. (c) The set of elements in all stacks contains an encoding of total and partial query answers.

At each step, PathStack traverses each element list sequentially, picks the next node (in document order) and removes from the various stacks elements whose ranges are before this node's range. The removed elements are guaranteed not to participate in any of the subsequent joins. At each point, the stacks contain elements whose ranges are either before or include the range of the current node, but have no ranges located after this range.

When a leaf node is accessed the algorithm outputs the results corresponding to the path terminated by this node. This is accomplished by a bottom up traversal of the node instances in that path, which at this point reside in the stacks. A node is joined with all its join partners before an ancestor node is considered. Hence the results are given in leaf-to-root order.

Figure 5 presents an example of the algorithm's execution. At the first and second steps, the algorithm accesses nodes `a(1,10)` and `b(2,5)`, and pushes them into their stacks (AS and BS respectively). Node `b(2,5)` points to node `a(1,10)`, as the latter is the lowest ancestor of the former in the current path. Asterisks denote which part of each input list has been processed. At the third step, node `c(3,4)` is accessed and pushed into its stack with a pointer to node `b(2,5)`. Since node `c(3,4)` is a leaf, the algorithm outputs the results that exist at the path of this leaf, i.e. tuples `a(1,10),b(2,5),c(3,4)`. It then pops `c(3,4)`. The fourth step (not shown) involves accessing and pushing `b(6,9)` and popping `c(3,4)`, while the final step involves pushing `c(7,8)` at its stack, outputting the tuples corresponding to this path and then returning.

Version Range PathStack. The input for the *Version Range PathStack* (*VRP*) algorithm is a path expression along with the version ids of the versions that are to be queried. One important difference from *PathStack* is that each input node may belong to more than one version. Hence, before creating a

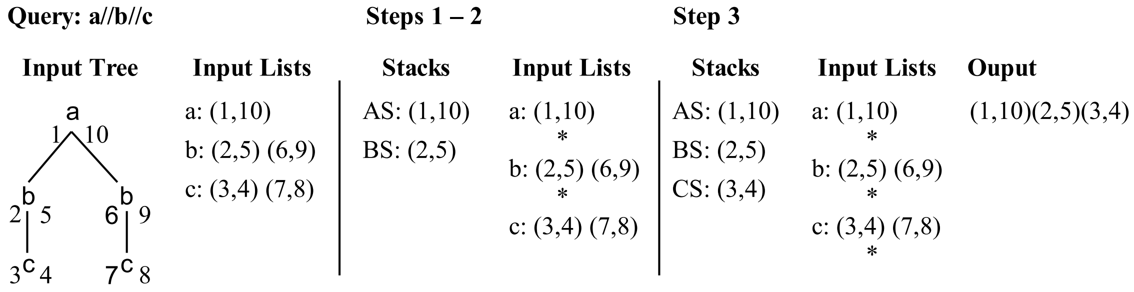


Figure 5: PathStack execution example.

path instance as a result, *VRP* must ensure that the nodes residing in that path are valid at the query specified versions. The algorithm assumes that all elements from a queried version exist in element lists in *left* number order. However in our environment, the various versions of an element list reside within a *BT-ElementList*. Hence, a layer between the *VRP* algorithm (called the *Element Stream*) and each of the input *BT-ElementLists* has to be implemented in order to provide the abstraction of element lists for the algorithm. This layer retrieves nodes belonging to the same version sorted by their *left* number. Its functionality is described below.

The sequence of records that reside in a *BT-ElementList* and belong to a particular version is called the *version list* for that version. At each step, the *Element Stream* keeps a pointer to the current element of each *version list*. There is one such pointer for each of the queried versions. At each step, the left positions of the current elements are retrieved. Subsequently, the one with the smallest left position is identified and returned. The traversal of each list is blocked until the current element is returned, and the pointer moves to the next record (Figure 6).

The invariant in the traversal procedure is that when a record of a version list is traversed, the other *version lists* access records with the same or larger left positions. Moreover, the way data are clustered on disk and traversed by the algorithm makes it possible to access each page only once for each version, instead of multiple times. If the number of queried versions is not large, pages can be pinned in memory when first accessed, and be there when subsequently requested by other versions. Even if the number of queried versions is large, chances are that the page requests are done simultaneously for each version, so no additional access overhead occurs.

Assume that the pages shown in Figure 4b correspond to the BT-ElementList for the a elements in the query of Figure 5. We describe the execution of *VRP* (in terms of pages and records accessed) with the same document input tree as in Figure 5 with the addition that the query requests versions 1, 2 and 3. In the first two steps, page 0 is accessed and each version list pointer returns the records with left position 1 and 2. In the third step, the *version list* for version 1 will access page 0 and return the record with left position 32, while the *version lists* for versions 2 and 3 will access pages 1 and 2 and return the records with left positions 32. The same pages will be accessed in subsequent steps until the algorithm terminates. As previously discussed, page 0, which has records shared by multiple versions is retrieved only once.

4 Experimental Evaluation

We generated synthetic data sets (XML document) from which we created 100 versions. Our XML document conforms to the very simple DTD where there is a root element under which there can exists multiple `a` elements, under which may exist multiple `b` elements, under which may exist multiple `c` elements. We used the expression `a//b//c` as our path query. Each version differs from the previous about 5%. Half of the changes are subtree additions, and the other half subtree deletions. Changes are uniformly distributed among data pages. Every version has two children versions. The page size is 8K and the generated element lists are approximately 11M. The buffer pool size is 100 blocks (approximately 2% of the input size). To generate the element lists from the document, we used an event based SAX parser. For the case of the BT-ElementList (BT-EL) the upper (lower) threshold was set to 90% (50% respectively).

```
mulitple_versions_getMinSource()

select position of non-blocked streams;
foreach non_bloced element
        bring next record r, in corresponding version
        if r.left < current_minimum
                current_minimum = r.left;
foreach version
        if current_record.left == current_left_minimim
                append record to the result;
                unblock corresponding stream;
return result
```

Figure 6: Next element retrieval in VRP.

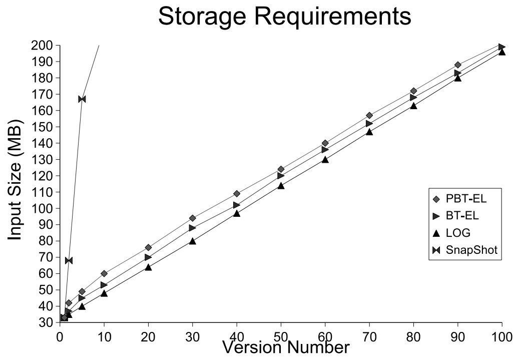
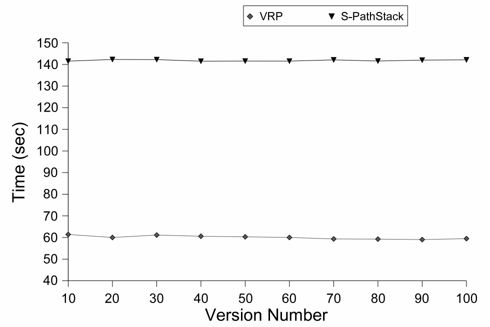


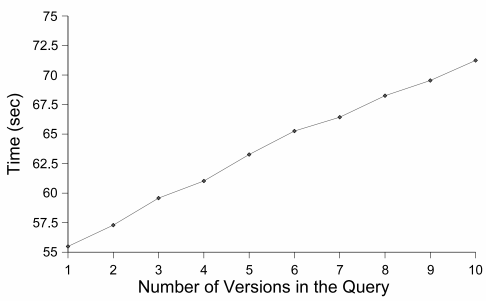
Figure 7: Storage Utilization.

Storage Utilization. Figure 7 shows the space required by the BT-EL and the minimal storage required by an append_at_the_end "log", which keeps the changes that occur in an element list (*log*). Note that such a "log" approach is rather impractical for query processing and serves only as a lower extreme for storage comparison purposes. This figure also shows the space used by the BT-ElementList without the optimizations we introduced in section 3.1 (*PBT-EL*). Clearly, *BT-EL* and *PBT-EL* have storage that is close to the minimal case. Among this two variations, *BT-EL* is more space efficient, because less replication related to a page overflow occurs (a smaller number of records is copied, so as the new page is not underutilized). Also, fewer page consolidations occur, since a page is labeled as underutilized less often. The other extreme (*snapshot*), where a copy of each element list is stored as a whole for each version, requires prohibitively large space. This is what a straightforward implementation of the original PathStack algorithm would require; after about 7 versions, its space is above 200MB (and falls outside the graph).

Scalability of Join Performance. In this experiment we investigated the effect of the archive size on the join performance. We fixed the number of versions in the query to three and kept adding versions to the archive, until a total of 100 versions. To measure the query time, we repeated the experiment for different groups of three versions; for each of them we run the algorithm ten times taking the average as the query time. Figure 8a shows the join performance comparison of the *VRP* (using the BT-ElementLists) against the original PathStack, which uses the *snapshot* approach (*S-PathStack*). The original PathStack algorithm works with one version at a time. Since the query requires three versions, the *S-PathStack* algorithm must run for each version separately (there is also one more pass at the end to avoid duplicates, but this is not counted in this experiment). The *VRP* algorithm has clearly the lowest join performance, independent of the size of the archive, and uses much less space.



(a) Join performance scalability



(b) Performance of VPR when varying the number of queried versions

Figure 8: Join Performance.

Moreover, we examined the query performance of a variation of *PathStack* implemented on the "log" approach (*L-PathStack*). Each element list has to be reconstructed from the "log" before PathStack is run, a cost that increases with the size of the archive. Even with 10 versions, the cost of the algorithm is 152 sec (not depicted in Figure 8a) and gradually increases to 174 sec with 40 versions. Clearly, this approach is prohibitive due to its large join processing time.

Join Performance with varying number of queried versions. Figure 8b shows the join (query) performance of *VRP* when we vary the number of versions in the query. The performance of *S-PathStack* is not shown as it increases rapidly with the number of queried versions (it is about double for two versions, triple for three versions, etc.) The performance of *VRP* increases much less rapidly than *S-PathStack* as expected. The reason is because the algorithm attempts to access the shared records only once. Clearly, *VRP* scales graciously with the number of queried versions as well.

5 Conclusions

XML has become the standard in supporting e-commerce and e-government applications. Typically such applications are collaborative in nature, which creates the need to support branched versions of XML documents. Being able to efficiently store and quickly extract the desired information from a multiversioned XML document is thus vital. We examined the problem of answering path queries over branched versions of an XML archive, considering also queries that access many versions at a time. We first proposed a storage technique to represent the versioned archive with a small replication factor. We then proposed the

VRP algorithm to answer path expression queries in the multiple version environment. We experimentally showed that the solutions described provide efficient and scalable performance.

Acknowledgements: This work was supported by NSF grants IIS-9907477, IIS-0220148 and UCdimi01-10122.

References

[1] S.Al-Khalifa, H.V.Jagadish, N.Koudas, J.M. Patel, D.Srivastava, Y.Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", *Proc. of ICDE*, 2002.

[2] J.P.Baron, M.J.Shaw, A.D. Bailey Jr, "Web-based E-catalog Systems in B2B Procurement", CACM, Vol.43, No.5, 2000.

[3] B.Becker, S.Gschwind, T.Ohler, B.Seeger, P.Wildmayer, "An Asymptotically Optimal Multiversion B-Tree", VLDB Journal, Vol. 5, No. 4, 1996.

[4] N.Bruno, N.Koudas, D.Srivastava, "Holistic Twig Joins:Optimal XML Pattern Matching", *Proc.of ACM SIGMOD*, 2002.

[5] P.Buneman, S.Khanna, K.Tajima, W.C.Tan, "Archiving Scientific Data", *Proc. of ACM SIGMOD*, 2002.

[6] S.Chawathe, H.Garcia-Molina, "Representing and querying changes in semistructured data", *Proc.of IEEE ICDE*,1998.

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

[8] S-Y.Chien, V.J.Tsotras, C.Zaniolo, D.Zhang, "Efficient Complex Query Support for Multiversion XML Documents", *Proc. of EDBT*, 2002.

[9] S.-Y.Chien, Z.Vagena, D.Zhang, V.J.Tsotras, C.Zaniolo, "Efficient Structural Joins on Indexed XML Documents", *Proc. of VLDB*, 2002.

[10] M.P.Consens, T.Milo, "Optimizing Queries on files", *Proc. of SIGMOD*, 1994.

[11] E.Cohen, H.Kaplan, T.Milo, "Labeling Dynamic XML Trees", *PODS*, 2002.

[12] T.Fiebig, G.Moerkotte,"Evaluating Queries on Structure with eXtended Access Support Relations",*Proc.of WebDB*,2000.

[13] G.Gottlob, C.Koch, R.Pichler, "Efficient Algorithms for Processing XPath Queries", *Proc. of VLDB*, Hong Kong 2002.

[14] H.Jiang , H.Lu, W.Wang, B.C.Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Join", *Proc. of ICDE*, 2003.

[15] L.Jiang, B.Salzberg, D.B.Lomet, M.B.Garca,"The BT-tree: A Branched and Temporal Access Method", *VLDB*, 2000.

[16] G.M.Landau, J.P.Schmidt, V.J.Tsotras, "Historical Queries Along Multiple Lines of Time Evolution", *VLDB Journal*, Vol.4, No.4, 1995.

[17] S.Lanka, E.Mays, "Fully persistent B+-trees", *Proc. ACM SIGMOD*, 1991.

[18] J.Leukel, V.Schmitz, F-D.Dorloff, "Exchange of catalog data in B2B relationships", *Proc. of ICWI*, 2002.

[19] Q.Li, B.Moon, "Indexing and Querying XML Data for Regular Path Expressions", *Proc. of VLDB*, 2001.

[20] D.Lomet, B.Salzberg, "Access Methods for Multiversion Data", *Proc. of SIGMOD*, 1989.

[21] A.Marian, S.Abiteboul, G.Cobena,L. Mignet, "Change-Centric Management of Versions in an XML Warehouse", *Proc. of VLDB*, 2001.

[22] G.Ozsoyoglu, R.T.Snodgrass, "Temporal and Real-Time Databases: A Survey", *TKDE*, Vol.7, No. 4, 1995.

[23] C.Quix, M.Schoop, M.Jeusfeld, "Business Data Management for Business-to-Business Electronic Commerce", *Sigmod Record*, Vol.31, No.1, March 2002.

[24] B.Salzberg, V.J.Tsotras, "Comparison of Access Methods for Time-Evolving Data", *ACM Computing Surveys*, Vol.31, No.2, 1999.

[25] M.Schoop, C.Quix, "DOC.COM: a framework for effective negotiation support in electronic marketplaces", *Computer Networks*, Vol.37, No.2, 2001, Elsevier.

[26] I.Tatarinov, et. al, "Updating XML", *Proc. of ACM SIGMOD*, 2001.

[27] Z.Vagena, V.J.Tsotras, "Path-expression Queries over Multiversion XML Documents", *Proc. of WebDB*, 2003.

[28] W.Wei, J.Haifeng, H.Lu, J-X.Yu, "PBiTree Coding and Efficient Processing of Containment Join" *Proc. of ICDE*, 2003.

[29] R.K.Wong, N.Lam, "Managing and Querying Multi-Version XML data with Update Logging" *Proc. of DocEng*, 2002.

[30] C.Zhang, J.Naughton, D.DeWitt, Q.Luo, G.Lohman, "On Supporting Containment Queries in Relational Database Management Systems" *Proc. of SIGMOD*, 2001.

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

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