Path-expression Queries over Multiversion XML Documents

Zografoula Vagena
Department of Computer Science
University of California, Riverside
foula@cs.ucr.edu

Vassilis J. Tsotras
Department of Computer Science
University of California, Riverside
tsotras@cs.ucr.edu

Abstract

In this paper we address path-expression queries on multiversion XML documents. Such queries are typically implemented on static (i.e., non-versioned) documents as path joins, using numbering schemes that maintain the structural relationships among the document elements. We extend previously proposed pattern matching techniques so as to support versions. We first present an easily updatable numbering scheme that efficiently captures structural relationships among the elements of the dynamically evolving document. We then propose a variation of Pathstack, an optimal pattern matching algorithm, that addresses the characteristics of our environment. Finally, through a thorough experimental evaluation we investigate two storage techniques in terms of space utilization and query efficiency.

1 Introduction

XML is gaining considerable attention as a standard for authoring, storing, exchanging and presenting documents [17]. In many applications, a document, as any other data source, may be subject to updates during its lifetime [13]. Such updates typically create a sequence of document versions making it possible for the users to query not only the most recent version but also previous ones. Moreover, users may be interested in querying the document evolution (i.e., *deltas*), too [12, 7, 6, 5, 4].

Given the tree structure of XML documents, path-expression queries represent the core of XML query processing [15, 16]. A path expression defines a series of 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 static documents [11, 1, 8, 3, 10, 14]. One common assumption in these approaches is that a (range-based) numbering scheme [11, 1] is embedded on the document tree in order to capture the above structural relationships among the document nodes. With this numbering scheme, path-expression queries are transformed to path joins [1, 3]. Holistic Joins ([3]) represent the state of the art in optimally processing path expression queries. Such joins utilize the document numbering scheme to identify in constant time, the relative position of any two nodes. Nevertheless, existing range-based numbering schemes are not easily updatable, a crucial issue when the document evolves (as is the case with versioning). While updatable prefix-based numbering schemes have been also proposed [9], they are not applicable to holistic joins. Moreover, prefix-based schemes may use variable length encodings and typically need more space than range-based numberings.

Previous work on version management of semi-structured documents [6, 5] and XML in particular [7, 12, 4], has considered various aspects of the storage and manipulation of multiversion documents. [7] proposed techniques that represent the document evolution as edit scripts or object references with the focus on fast reconstruction of any particular document version. [12] considers queries on the XML document's evolution changes, while [4] concentrates on the compact storage of a XML archival system, while its version querying assumes the existence of key identifiers.

Nevertheless, none of these approaches is optimized for path-expression queries in a multiversion XML environment. This is a novel problem since it requires easily updatable numbering schemes to capture the structural characteristics of an evolving document. Moreover, our solutions will offer the ability to query any document version at different granularities. More specifically, we devise algorithms to efficiently answer queries that specify patterns of selection predicates on multiple elements that have some specified tree structure relationships, over particular document versions.

The rest of the paper is organized as follows. Section 2 presents our techniques while section 3 contains the experimental evaluation. Finally, section 4 concludes the paper and offers directions for further research.

2 Structural Joins on Document Versions

A straightforward way to use holistic joins on versioned documents is to renumber the whole document at every update. We first describe a document representation for which no document renumbering is needed in the presence of updates (section 2.1). This enables the utilization of holistic joins (with minor changes). The updated algorithm is presented in section 2.2. Moreover, we describe a physical organization of document versions that better preserves the logical order of the document elements (section 2.3).

2.1 Representing regions and elements in an XML document

Previous approaches use numbering schemes that map the problem of checking for the ancestor-descendant relationships to the one of checking for range inclusion. Every node in the document is enhanced with two additional numbers that create a range. To check structural relationships between any two nodes one has to check the corresponding ranges: one node is descendant of another if and only if the range assigned to the former is included in the range assigned to the latter. Those schemes face the problem of range shortage, when updates happen in the document. Sooner or later there will be a moment where the numbering scheme for the whole document must be reapplied.

In [12] a XID-map is used so as to create persistent object identifiers for each document node. An XID-map provides a mapping between nodes of a tree and some integers. We utilize a similar scheme so that to represent the relative positions between document parts.

To mitigate the problem we propose to represent a document with different granularities. We cluster the elements of the documents into regions defined by the updates occurring in the document. Originally, each region is a subtree that is added or deleted from some part of the document. Updates on the document may split that region into subregions. The structural relationships of the elements within the same region are identified using ranges in the same way as in the previous case, while elements that belong to different regions are identified by the structural relationships of the regions that contain them. Certain elements play eminent role in the identification of the inter-region relationship. These are the boundary elements, i.e. the last element of each region. Moreover, there exists a document-map, which represents the relative places of the regions.

Document Evolution Originally the whole document defines only one region. The document map contains one range, the range of numbers that are used to number this region. The numbers within a region don't need to be consecutive, any of the proposed range based numbering schemes can be employed. The only constraint is that the range of numbers that one region uses cannot be subrange of any other range, as well as the ranges should not overlap. To achieve that, the largest number so far is retained, so that new region numbering start from this number.

When a new subtree is added, a new region is created. The region is numbered starting with the current next available number and a new range, the one defined by the numbers employed in the region, is constructed. The element under which the new subtree is going to be added is identified and its region is split into two new subregions, the one starting from the start of the old range and ending at the left position of the element and the other starting from the right position of the element and ending at the end of the old range. This change is reflected at the document map, by the replacement of the old region range with the two new ones. Moreover, the range of the added region is placed between the created subranges.

Deletion of a subtree might delete its range from the document map, if the subtree defined a whole range. Moreover, it might lead to region merge, if an old region is reformed.

Because we present the document map using ranges, the space occupied by the former is small and can be kept in main memory. However, if it becomes to large, we can migrate it to the disk, without loss in the algorithm performance, as will be shown in section 2.2.1.

Figure 1 shows the the evolution of a document, as well as the document maps for each of the document versions. We assume w.l.o.g that a new update triggers the creation of a new document version. In the figure deletion of a subtree leaves the document map intact. The reason is because what is deleted is not a whole region.

Before continuing we need to establish some notation. We call the node under which a new subtree is added the root node of the region corresponding to that subtree. Moreover, we call the region within which we add a new region the parent region of the latter. We define the ancestor-descendant relationships between regions recursively.

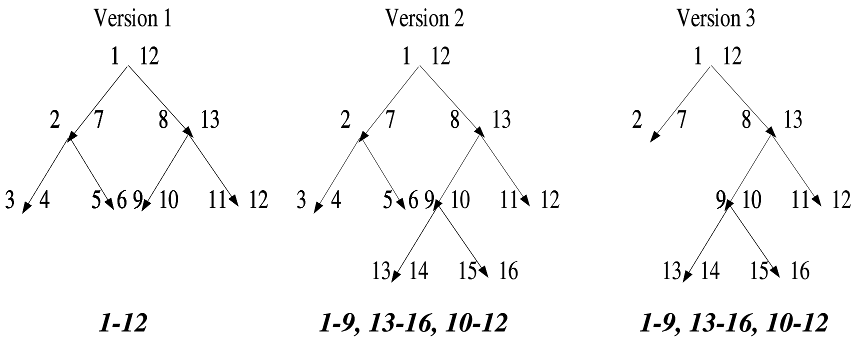


Figure 1: Document Evolution.

2.2 Path Queries over a particular version

Using the numbering scheme described above one can perform the structural join over a particular version by extending existing algorithms so that the latters take into account the document map. In this section we extend *PathStack*, an optimal structural join algorithm for path queries. We begin by providing a brief description of the algorithm. Then, we report the changes that need to be applied to get the desired results.

2.2.1 PathStack

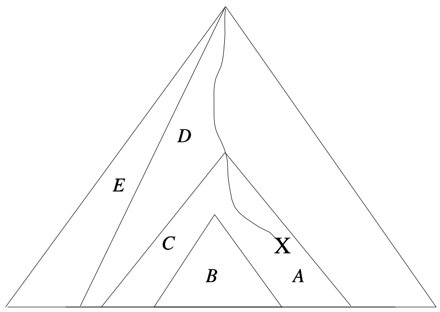
PathStack receives as input the element lists of the element types that participate in the query. An *element list* is the list containing all the elements of the same type, enhanced with the appropriate numbering scheme (left, right position). Moreover, every query node has a stack associated with it. Each element in the stack points to an element in the stack of the ancestor node as the latter is defined by the query. The computation invariant is that (a) each stack contains document nodes that belong to the same path, in descendant order and (b) the set of stacks nodes contains an encoding of total and partial answers. At each step, the algorithm traverses each element list sequentially and picks the next node (as defined by the inorder traversal of the document) and removes from the stacks nodes whose ranges are before the current range (as the formers are guaranteed not to participate in any of the subsequent joins. The way the algorithm chooses the next element guarantees that the stacks contain elements whose ranges are either before the current range or include the current range (i.e. ancestor elements) but never ranges located after the current one.

2.2.2 Version PathStack

In Figure 2 we provide the updated *PathStack* algorithm which operates on a document version. The inputs for the algorithm are the element lists as in the previous case, along with the document map that describes the current version. Each list is constructed as follows: for the first version of the document the list contains all the elements of the same type, enhanced with the numbering scheme and sorted on left position, as in the previous case. When new subtrees are added, their elements are numbered and are appended in their corresponding lists, again sorted. Subtree deletions have no affect over the lists. To be able to access different parts of the lists on will we index the formers with an index that supports range queries (e.g. a B+tree). The index will allow to access the document regions in the sequence implied by the document map. Specifically, it returns the smallest element in the list which resides between two given elements (the region boundaries).

The algorithm is very similar as the original *PathStack*. Again, at each step the algorithm picks the next element, as this is defined by the sequence of regions within the document and the inorder sequence of each element within the list the former belongs. To find the next element, the current element of each input stream is checked and if it is past the end of the current region, the B+tree is probed so as find the portion of the list belonging to the first subsequent region. After finding the next element one has to clear the stacks from the elements that are not going to participate in subsequent joins. In this case the algorithm has to take into account the relative position of the regions. Figure 2 shows the possible relative position between the current element and the current top of a stack. Specifically, the current current top element can:

```
Algorithm Version PathStack
while not_end_of_input
  find next element e
  find range where element belongs
  for every stack
    if top_stack is not ancestor of e
      pop top_stack
move e to its stack
if is_leaf(e)
    showSolutions
    pop(e)
```



Relative positions between current and stack elements.

Figure 2: Version PathStack.

- be an ancestor of the current element
- belong to the same region as the current element and its range is before the range of the current element
- belong to a region inside the current element's region, which is located at the left of the current element, or any other region created inside it
- belong to a region having the same parent region as the current element one and exist at the left of the current element
- belong to a region that exists at the left of an ancestor of the current element's region.

In all the above cases, except the first one, the current top stack element has to be removed from the stack. It is easier to check whether the current top stack element is ancestor of the current element. The way to achieve it is to check whether the top stack element is ancestor of the root of the ancestor region that splits the top stack element's parent region. If not then the stack element is removed from the stack. This can be performed efficiently even when the map resides on the disk by enhancing each region with a scheme to identify structural relationships between regions. For this case we can use an extensible one like a suffix numbering schemes. Having that we identify all the root elements of the regions that have split the current top stack element region and if any of them is descendant of the element we check whether it is ancestor of the region containing the current element.

2.3 Making the element lists persistent

In the previous subsection one copy of each element list is kept for all the versions and the document map is used to access the desired parts of the list for each version. New regions are appended at the end of the list. This leads to poor physical clustering of the elements and as a result poor retrieval performance. Precisely, one might need to get the same page from disk over and over, depending on which parts of the list are stored there. It would be beneficial if the lists were created in such a way, so that the logical sequence of each element is preserved as much as possible. We propose a solution that, for each version, (a) retrieves only pages that are useful, i.e. contain enough fields that are valid for the version and (b) the fields within the page follow the logical order of the element list. To achieve that we keep the list of useful pages in logical order for each version. Moreover, we employ split and merge techniques similar to the ones used in the multiversion B-Tree ([2]); after we find the element in the element list after which we will insert the new region we check whether there is any space left in the current page. If so, we insert the field. Otherwise we check whether we can copy only the useful records for the current version in a new page. And replace the old page with the new one in the list of useful pages for the version. To do that we perform an artificial removal of the alive records in the old page, and their insertion in the new page. If all the records in the page are alive, we split the page into two new ones that replace the former for the current version. While records are removed the page can become underutilized. In this case we copy valid records of the page into a new one, making the old one non-useful.

The changes that have to be done in the *Version PathStack* so that the former works with the new storage scheme are minimal. Specifically, while accessing each list in its physical order to find the new current element for the algorithm, we need to keep track which document region the list pointer accesses at each time. To do that we can sequentially search the document map, or in the case where the former is too large to fit in main memory we can employ a content index (like B+-Tree) to speedup the search.

3 Performance Analysis

We compare the performance of three algorithms, namely the original *PathStack* algorithm (we call it *PathStack* from now on, the *Version PathStack* using the B+-tree (we call it *BPathStack* from now on) to index the element lists and the *Version PathStack* making the element lists persisten (we call it *SPathStack* from now on). In the first group of experiments we compare the last two algorithms in terms of storage utilization and the second group of experiments we compare all of them in terms of join performance. Our aim is to compare the performance overhead of our techniques compared to having the one we would get if we could have a snapshot for each one version, in which case *PathStack* would be optimal. We have generated synthetic data sets (XML document) from which we generated 10 more 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 100%. Half of the changes are subtree additions, and the other half subtree deletions. The changes are uniformly distributed among data pages. The page size is set to 8K and the generated element lists are approximately 11M. The buffer pool size was set to 100 blocks (i.e. approximately 2% of the input size). To generate the element lists from the document, we used an event based SAX parser. For *PathStack* we parsed the whole document for each new version. For the case of persistent lists we set the usefulness threshold to 50%, i.e. whenever 50% of the records in the page are cease to be valid for the current version then all the valid records are copied to a new page.

3.1 Storage Utilization

The left part of Figure 3 shows the space occupied by the input (element lists) along the versions. Persistent list is more space efficient than the B+tree as the latter incurs the cost of the storing the index pages, along with the added nodes, while in the case of the persistent list the storage is on average proportional to the added nodes. We expect the difference between the two schemes to decrease as we increase and probably gets inverted, as page invalidations will incur more replication.

3.2 Join Performance

In the right part of figure 3 we show the performance of the three algorithms. The performance of the *PathStack* is almost the same along the different versions, as the changes we perform don't affect the input size and the join selectivity a lot. The performance of *PPathStack* is slightly larger than the *PathStack*. This is attributed to the fact that the size of the lists per version is slightly larger as a result of page invalidation and page copies. However, the execution of *PathStack* is not possible without the storage of the whole list per version as well as the renumbering of the document, two requirements difficult or even impossible to satisfy in practice. *BPathStack* performs worse than both algorithms. This is attributed to the fact that the algorithm has to access parts of the input lists in a different order than their physical storage (something which is exacerbated with the persistent list were the elements within the page have the same physical sequence as the algorithm is going to access them). Plus *BPathStack* has to incur the overhead of the index traversal.

4 Conclusions & Future Work

In this paper we deal with the problem of answering path queries over particular versions of an XML archive. We propose a representation of the document that reveals structural relationships between document nodes and adapts gracefully when updates occur. We then extend existing optimal algorithms for the path query problem so that the latters take into account the new representation of the document, making it possible to answer the same queries on different versions. Furthermore, we compare two storage techniques for each element list. We presented the techniques

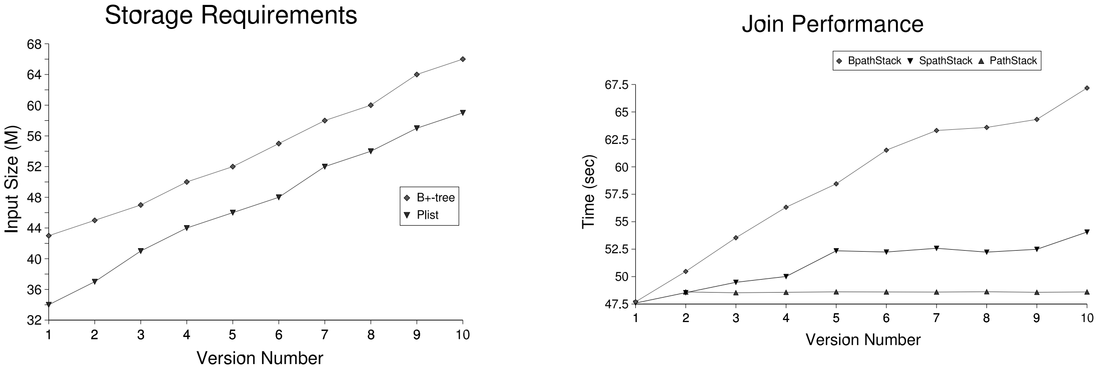


Figure 3: Performance Results.

in the case of path queries, but the latters can be adapted to work for branch queries also. As part of our future work we are planning to investigate the problem of querying the structural evolution of the document in different granularities.

References

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

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

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

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

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

[6] S. Chawathe, S. Abiteboul and J. Widom, "Managing Historical Semistructured Data", *TAPOS*, 5(3):143-162, 1999.

[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, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents", *Proc. of VLDB*, Hong Kong, China, 2002.

[9] E. Cohen, H. Kaplan and T. Milo, "Labeling Dynamic XML Trees", *Proc. of PODS*, 2002.

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

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

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

[13] I. Tatarinov, Z.G. Ives, et. al., "Updating XML", *Proc. of ACM SIGMOD*, 413–424, 2001.

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

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

[16] World Wide Web Consortium, "XQuery 1.0: An XML Query Language", W3C Working Draft Jun 7, 2001 (work in progress). http://www.w3.org/TR/xquery/

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