Experimental Evaluation of Query Processing Techniques over Multiversion XML Documents

Adam Woss
Computer Science
University of California, Riverside
awoss@cs.ucr.edu

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

ABSTRACT

Various approaches have been recently proposed for storing the evolution of an XML document, thereby preserving useful past information about the document and thus the ability to query it. While the importance of maintaining a document's evolution has been recognized, relatively little research has been done on how to adapt efficient XML querying techniques in a multiversion environment. Such methods would allow the user to efficiently perform structural queries over any past version or interval of versions of the XML document. In this paper, we examine query processing techniques on (linear) multiversion XML documents. Specifically, we propose efficient modifications to existing XML query processing algorithms and compare them to more traditional approaches. Our experimental results demonstrate the advantages of the modified algorithms.

1. INTRODUCTION

XML has emerged as the standard for publishing semi-structured data and exchanging it over the Internet. Interestingly, the large majority of XML documents that are disseminated undergo modifications (e.g additions, removals and updates) over time [7]. These modifications, in effect, create multiple versions of the XML document as time progresses. In many cases maintaining the past versions is required as it provides the ability to search over historical information. Multiversion support for XML documents is also needed in several applications, like software configuration, cooperative authoring, web document warehouses etc. Consequently, devising an effective solution to storing multiversion XML documents has attracted a good deal of research interest over recent years [1][5][15]. Most approaches assume a linear versioning scheme: when a document is updated a new version is created, emanating from the last version.

Nevertheless, relatively little research [19] has been performed to adapt current efficient XML querying processing techniques (in particular "holistic" ones like Twigstack [10] and LCS-TRIM [14]) to query multiversion XML documents. Instead, existing techniques either focus on converting XML documents to relational temporal tables [5] (and use SQL-like querying) or use edit scripts [1, 15] to reconstruct the XML document for a given version.

When considering non-versioned data, holistic processing methods have been shown to outperform the relational approach; it is thus possible that, if well adapted, they will hold the same promise for versioned data as well. Moreover, the use of edit scripts for versioned XML is not optimized towards complex structural queries that ask for part of the document. More related is the work in [19] which however has focused on simple path queries using an extension of Pathstack [10]. Here we built on that experience and examine the more complex twig structural queries, using both a merge-join [10] and a subsequence matching [14] approach. Twig support is important since twigs are considered as the building block for XML querying languages (like XPath and XQuery): queries typically specify patterns of selection predicates on multiple elements that have some specified tree structure relationships.

2. BACKGROUND

An XML document is usually modeled as a rooted, ordered, labeled tree. Each node in the tree corresponds to an element or a value, and the edges represent immediate element-sub element or element-value structural relationships. When considering structural XML queries, one has to quickly determine parent-child and ancestor-descendant relations for a given set of tree nodes. This is achieved by using special labeling schemes that capture such relationships between node pairs without having to traverse the path to each node. An important feature of a numbering scheme for multiversioned XML documents, is how the scheme adapts to changes. We thus considered various such schemes that are described in Section 2.1. Moreover, in Section 2.2 we discuss previous approaches on maintaining and querying multiversion XML documents.

2.1 Labeling Schemes

One of the first XML labeling schemes used an interval-based [10] approach. Each node is typically represented by the following tuple (DocId, LeftPos : RightPos, Level). Here: (i) DocId is a unique identifier for the node's document; (ii) LeftPos and RightPos represent the preorder and postorder traversal of the node in the tree. This interval also contains the intervals of a node's tree descendants; (iii)

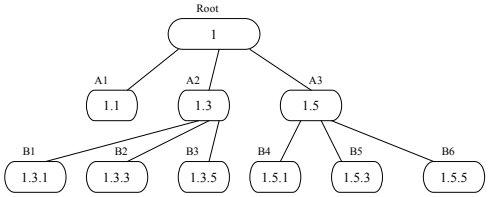

Figure 1: ORDPATH labeling scheme

Level is the depth of the node. Given two tree nodes E_1 and E_2, with labels $(D_1, L_1 : R_1, L_1)$ and $(D_2, L_2 : R_2, L_2)$ respectively, one can quickly determine if E_2 and E_1 represent an ancestor-descendant relationship if $L_1 < L_2$ and $R_2 < R1$. By also ensuring that $L_2 = L_1 + 1$ we can infer that E_1 and E_2 are specifically a parent-child relationship.

However, interval-based labeling is not well suited when frequent updates are made to the document. When a new node is added to a document tree, the exact range which should be used for the LeftPos and RightPos is unknown, thus it becomes necessary to re-label the entire XML document. Such relabeling is not a practical option when considering multiversioned documents since updates are often; moreover, such relabeling could also affect any already built indexes that use the labeling scheme.

The prefix labeling scheme [7], uses the prefix of a label to determine if an ancestor-descendant relationship exists. Prefix labeling requires no re-labeling if the order of the XML tree is not of concern, but if order is of importance, re-labeling will be required. Also, as the document grows in size there is a significant storage overhead associated with the prefix labels. Furthermore, the comparison of prefixes to establish structural relationships is less efficient then the integer comparisons used by the interval-based approach.

Prime numbers have been used in [17] to maintain tree order without any re-labeling. Nodes are numbered by the product of the "parent-label" (inherited from the parent node) and the "self label" (assigned by the labeling scheme). A new prime number is assigned as the "self-label" of a newly inserted node. Although prime labeling supports updates, the size of labels will become large as subsequent inserts require unique "self-labels". Moreover, re-calculation is both necessary and costly in order to determine structural relationships.

Lastly, we considered ORDPATH [12] which is a Dewey variant [9]. ORDPATH is a hierarchical scheme which improves upon the prefix methods. It reserves even and negative values for subsequent inserts into the XML tree, which in turn allows for order to be maintained without re-labeling. For example, in Figure 1, if a new node were to be inserted between nodes 'A2' and 'A3' it would be given the ORDPATH label '1.4.1'. This method of inserting is called "careting in" and if it is applied frequently it could increase the length of labels significantly. However, [12] showed that excessive "careting in" is rare in practice. Based on its practicality and updatability, we decided to implement our multiversion solutions using the ORDPATH numbering.

2.2 Maintaining Evolving XML Data

Temporal (relational) databases have been proposed [2] for maintaining evolving data. When considering XML and semistructured data, one approach is to transform the data into relations and then use a temporal database approach. This however requires (sometimes complex) extensions to both the database and SQL. Instead, we focus on more XML-conscious approaches (i.e. "native") designed specifically for archiving changes undergone by XML documents.

The approach in [1] maintains a change-centric representation of the XML data, by focusing on the changes themselves via the use of deltas or edit scripts. It relies on its ability to track XML nodes through time using persistent identifiers called XIDs. These XIDs are contained in the deltas, which allow them to easily recreate the document for a given version by scanning deltas and applying the changes to the mapped nodes. Although this approach easily and efficiently maintains all the changes made to a XML document, it may not be as efficient for structural XML querying because: (i) the XIDs are essentially a very primitive labeling scheme and as such do not provide any insight into ancestry relationships, (ii) there is a significant cost that may be incurred with having to reconstruct the document using deltas.

Another approach relies on clustering the history of each node on the node itself by maintaining a set of version-stamps [11]. This can efficiently support the retrieval of any specific version as well as provide the past history of a given tree node. However, much like the delta approach, it cannot efficiently support structural queries since it cannot check structural relationships between nodes without traversing the tree. Moreover, it does not maintain order among document nodes.

The referenced-based version model [15] aims to solve the limitations of edit-based approaches. In particular, a separate view is created for every version, but there are references that point to the maximum unchanged subtree in the previous version. In other words, the unchanged elements are shared among the subsequent views. [15] shows that the reference-based model has both improved storage and retrieval cost compared to more traditional approaches. However, the management of the views could lead to increased overhead as more and more versions are created; moreover, this approach was optimized towards full subtree version reconstruction.

The work in [16] addresses simple path expression queries (i.e., the special case where an element cannot appear in the subtree of an element with the same tag) in a multiversion XML document as combinations of partial version retrieval queries. [19] addresses path expression queries on a multiversion XML document using a region-based numbering scheme. Nevertheless, none of these approaches is optimized for general structural (twig) XML queries in a multiversion XML environment.

3. MULTIVERSION TWIG QUERIES

When considering the traditional non-versioned environment, finding all occurrences of a twig pattern is a fundamental operation for XML query processing. Original methods have focused on decomposing the query into multiple predicates and then merging the results; this may create false positives

that need to be eliminated in a second phase. Instead, recent work has focused on "holistic" processing techniques: a global matching of the query pattern is performed. Holistic matching has been shown to be superior in performance [10][14][13][6]. Holistic techniques can be broadly categorized into two differing approaches [8]. The first relies on merge-joins [10] while the second on subsequence matching [14, 13, 6].

Among the holistic methods, we consider further two representatives, namely, Twigstack [10] and LCS-TRIM [14], as they are currently the most efficient approaches in each of the two categories. Twigstack [10] creates a list T_q (called element list) that stores all nodes (elements) in the XML document of the same tag q. Such appearances of q are kept in T_q sorted by their LeftPos. When a twig query is processed, only the lists of tags that appear in the twig query are accessed. Also, each query node q is associated with a stack S_q. These stacks enable the compact encoding of (a possible larger number) intermediate partial results found while processing the element lists. Using the interval-based numbering scheme, the structural constraints are converted to joins. The algorithm can guarantee worst-case performance linear to the query input and the size of the query result. Moreover, the algorithm works for both sorted and unsorted twigs.

The LCS-TRIM [14] transforms both the XML document and the twig query into sequences (using Prufer encoding [13]) and then uses a dynamic programming approach to find the longest common subsequences (LCS) and thus all matches of the query. Equally instrumental to its performance is the novel structure-matching algorithm used to prune false positive matches. LCS-TRIM, however, is not particularly well suited for unordered matching as the sequences are constructed from tree traversals and thus, preserve order. Enumerating the set of all possible sequences of a query is at worst exponential in size. However twig queries are typically small and hence processing all possible variations of an unordered query will not be expensive in practice.

We now focus on extending these holistic matching algorithms to operate on multiversion XML documents. The major challenge is that current holistic techniques were developed with a static XML data model in mind. We propose two approaches that adapt the LCS-TRIM and Twigstack algorithms for use on multiversion XML documents. They both facilitate a multiversion B-tree index (MVBT [4]) that allows efficient access to the needed past versions (of the element lists, or the document sequence). We also consider a third approach that applies LCS-TRIM on a temporally clustered XML document. The design and implementation of the updated, multiversion query processing techniques is described next.

3.1 MVBT-Twigstack

As the name suggests, this algorithm modifies the original Twigstack [10] algorithm to facilitate a MVBT index on each element list. This MVBT effectively provides access to any past version of a given element list. The reduction of the multiversion XML problem to the use of MVBTs follows. Consider an XML document at version 1. Assume that it

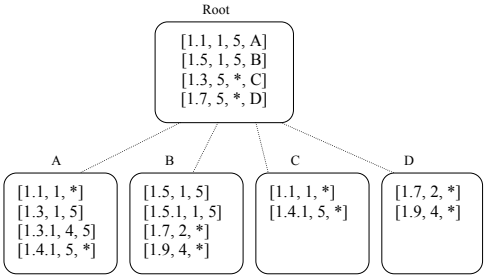

Figure 2: MVBT on element list

is numbered using the preorder/postorder traversal. Based on its LeftPos, each node q of the tree is assigned to an element list T_q. Conceptually, list T_q at version 1 is an ordered set of LeftPos numbers. To address a twig query for version 1, Twigstack needs to access T_q as is in version 1, in increasing LeftPos order. Now consider the same document at version 2. Assume that version 2 is created from version 1 by adding some new nodes in the XML document tree and deleting some others. All the node additions (or deletions) are translated into adding (resp. deleting) their positions in the appropriate element lists. Since the ORDPATH numbering guarantees that the existing nodes do not change their numbering (i.e. LeftPos) the addition of a new node is simply adding a LeftPos number in the appropriate list while maintaining order. Assume that the LeftPos order in a given element list is maintained by a B-tree. Then the evolution of this element list over subsequent versions can be maintained by making this B-tree partially persistent, i.e., a MVBT [4]. Consider a B-tree that went through an evolution of n versions and assume (for simplicity) that each version makes a constant number of updates. Then reconstructing the i-th version of the B-tree takes time $O(log_b n + s/b)$, where s is the size of the B-tree in version i and b is the page size in records; moreover the MVBT uses space $O(n/b)$ [4].

Using MVBTs for implementing the multiversion element lists provides the following advantages: (i) efficient access to the elements of a given version in increasing LeftPos order, and, (ii) coupled with ORDPATH labeling dynamic updates can be processed on the most current version of the list.

The MVBT-Twigstack algorithm still needs to process the nodes in the input element lists, but it no longer needs to sequentially scan the list from version 1. Instead, if the query is about version i, the starting position for each element list at version i is found by traversing the list's MVBT, which effectively prunes any irrelevant nodes (that were valid for older or later versions than version i). Consider Figure 2 which depicts a (small) MVBT over an element list. Each list record contains the ORDPATH number and a version interval; for example, record (1.1, 1, *) in page A, corresponds to an element that has ORDPATH 1.1, and was inserted in the list at version 1 (the '* implies this record is still valid). Similarly, element with ORDPATH 1.5 in page B was inserted in version 1 and deleted in version 5. When a leaf page reaches a given threshold the active nodes are copied to a new page, which is why some of the nodes found in pages A and B are also shown in pages C and D (details appear in [4]. The index records (as in the Root node) also contain pointers to children nodes. A query asking for the list at version 6 would direct the search from the Root to

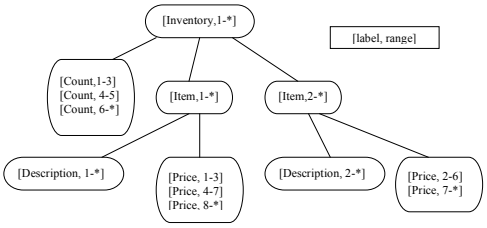

Figure 3: XML data model used with TLCS

pages C and D, thereby skipping any irrelevant data nodes in pages A and B. Much like the optimized TwigstackXB [10] we also maintain a pointer to the active page in our MVBT. This pointer allows us to advance to the next page or drill down further into the tree, effectively allowing us to adjust the level of granularity at which we skip records not included in the query.

3.2 MVBT-LCS

Our second approach modifies LCS-TRIM [14] to incorporate an MVBT. Instead of organizing tags into their corresponding element lists, the LCS-TRIM approach uses subsequence matching over the whole XML document Prufer sequence, which is created by a preorder traversal of the document, in LeftPos order. Hence another way to support multiversion twig queries is to efficiently maintain the versions of the XML document Prufer sequence.

Because of the ORDPATH numbering, a node addition in the XML document is added in a specific place in the ordered sequence. Thus the document evolution is reduced to maintaining the ordered sequence evolution. An MVBT can be used, allowing for efficient access to only the needed versions of the document sequence. Effectively, this allows for the pruning of non-relevant nodes (i.e., from other versions) without examining them first. After the MVBT has been traversed and the nodes of the sequence as of a given version are found, the modified longest common subsequence algorithm can be applied against the query subsequence.

3.3 Temporal LCS

A characteristic of the MVBT is that it provides efficient access to a particular version (whether this is an element list or the document sequence). Another organization of temporal data is to cluster object versions together, similar to the approach in [11]. This would not work well with Twigstack since it requires the element lists in order. Nevertheless, it could be used with LCS, in an approach we term as Temporal LCS (TLCS). Here each node in the XML document tree maintains a version interval which represents the versions for which this node was considered valid. Figure 3 is an example of the version-stamp based tree used for TLCS. Unlike the MVBT-LCS approach there is no way to efficiently construct the sequence as of a given version (as required by LCS-TRIM). Such computation adds an extra pre-processing step incorporated into TLCS to eliminate irrelevant nodes.

4. EXPERIMENTAL EVALUATION

We proceed with the experimental evaluation of our modified algorithms: MVBT-Twigstack, MVBT-LCS and TLCS. As a baseline for our experiments we also include two straightforward approaches, termed as "log-based" and "snapshot". The log-based approach simply adds the various changes occurring in a document, stamped by the version id when they occurred. This approach presents the smallest space consumption (linear to the number of updates). The snapshot approach stores the document tree as it is on each version; it thus uses large space (quadratic to the number of updates) but provides the fastest access to a particular version of the document.

ID	Query Expression
Q_1	//item_id[//location="United States"][//payment="Credit Card"]
Q_2	//region="europe"/item_id[//quantity="5"][//payment="Cash"]
Q_3	//item_id[//name][//payment][//description][//quantity][//location]
Q_4	//item_id[//mail/date="10/10/2000"][//payment="Credit Card"]
Q_5	//item_id/description[//shipping]

Table 1: Twig queries for XMark data set

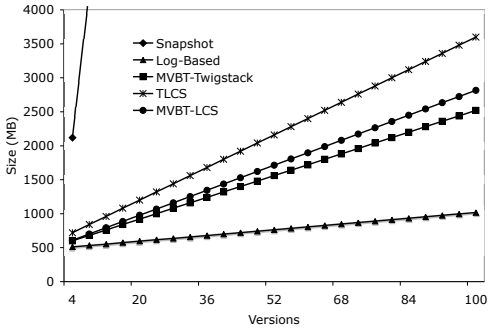

Figure 4: Storage Cost

All the experiments were run on a 2GHz Intel Core Duo with 4GB of main memory. We used the XMark [18] benchmark to generate the synthetic data sets for our experiments. The Xmark generator models data from that of an online auction, however, the data set generated was not a multiversion document. A python script was used to simulate new versions by applying a batch of inserts, removes and updates to the tree. In particular, the data set generated has a size of 500MB and just over 6 million tree nodes. A total of 100 document versions were created. Between versions about 5% of the current version was modified (by inserts, deletes, updates).

Table 1 represents the XPath queries used in the experiments. Each query is appended by a single version or an interval of consecutive versions so as to access the multiversion XML document. In particular, we considered a "small" version interval of five consecutive versions and a "large" interval of twenty consecutive versions. As for the query characteristics, both Q_1 and Q_5 are basic twig queries with low selectivity, while Q_3 and Q_4 contain a higher fan-out and depth causing slower runtimes.

4.1 Storage Cost

Figure 4 depicts the space usage as the XML document evolves over time. The log-based approach uses the minimal space, since there is no extra overhead associated with storing changes. Although the snapshot approach is able to retrieve a specific document version quickly, its space consumption as the number of document versions increase, clearly makes this approach impractical. TLCS, MVBT-

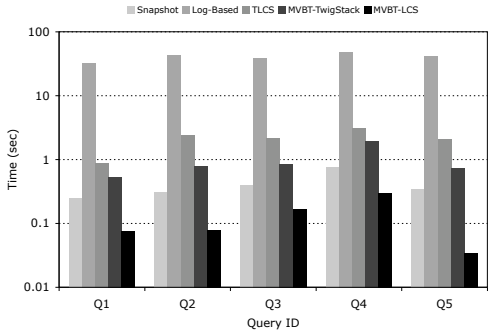

Figure 5: Ordered: Single Version

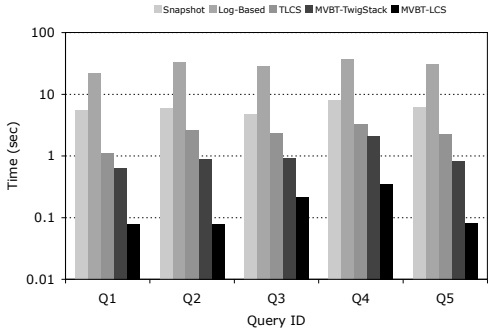

Figure 6: Ordered: Version Interval = 5

LCS, and MVBT-Twigstack maintain space that grows *linearly* with the total number of changes made in the evolution.

4.2 Query Time

We distinguish between ordered and unordered matches. The subsequence based methods need to run all possible orderings of an unordered twig query so as to find all unordered matches.

4.2.1 Ordered Matches

Figures 5-7 illustrate the execution time (note the logarithmic scale) of $Q_1 - Q_5$ for the three version ranges, respectively. The single and interval (5 or 20 consecutive) versions were chosen randomly within the document's evolution. The Snapshot approach could be implemented using either LCS or Twigstack. In these figures we report the Twigstack implementation (effectively, for each version, the snapshot of each element list is stored and accessed by the original Twigstack algorithm). For a single version query, the Snapshot approach performs well. However, as the version range increases the performance quickly degrades; this is because results from each document version are first collected and then merged. While the Log-Based method provided the minimal amount of storage space, its query runtimes are too expensive regardless of version range. This is attributed to the overhead incurred when having to recreate the document for a given version.

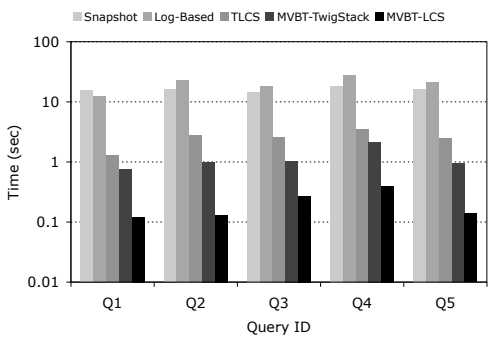

Figure 7: Ordered: Version Interval = 20

TLCS is consistently worse than both the MVBT-Twigstack and MVBT-LCS approaches. This is because the MVBT-based approaches focus the algorithms to the nodes valid for the version(s) in the query. Instead, the TLCS has to parse the overall document sequence (including nodes that are not related to the query version(s)) before it creates the document sequence needed for the query. The TLCS processing time is unaffected by the size of the version range since the processing is effectively the same. In contrast, MVBT-Twigstack and MVBT-LCS processing increases as the version range increases since the number of nodes accessed by either MVBT-Twigstack or MVBT-LCS also increases.

Among the MVBT based approaches, the performance of MVBT-LCS is faster than MVBT-Twigstack. This observation is similar to what has been reported for ordered twig queries in the non-versioned environment [14]. Among the various queries, MVBT-LCS shows the smaller runtime for queries Q_1 and Q_5; this is to be expected since these queries have the lowest selectivity.

The support of versions through the use of the MVBT has a relatively small overhead on the traditional query processing algorithms. This can be seen in Figure 5, when comparing the Snapshot approach with MVBT-Twigstack for a single version query. The Snapshot approach used Twigstack on the stored version of the document, while MVBT-Twigstack has the overhead of using the MVBT. We observed the same when comparing LCS-TRIM with MVBT-LCS for a single version [3].

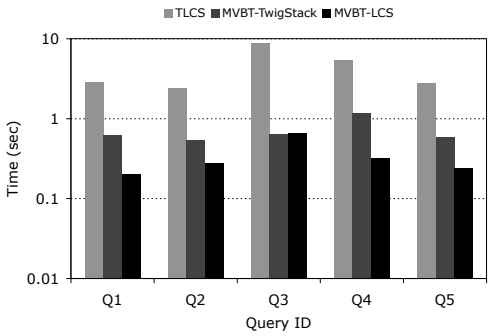

Figure 8: Unordered: Single Version

4.2.2 Unordered Matches

Figures 8-10 depict the execution times of $Q_1 - Q_5$ when unordered matches are desired. For simplicity the graphs show only the TLCS, MVBT-Twigstack and MVBT-LCS. The MVBT-Twigstack behavior is similar with the ordered case, since the Twigstack approach can easily do both ordered and unordered matchings. Instead, the LCS based approaches create an ordered sequence for each twig query. As a result, all configurations of a query must be processed for finding the unordered matchings (more twigs need to be

processed per query). As expected the runtime of the LCS approaches increases.

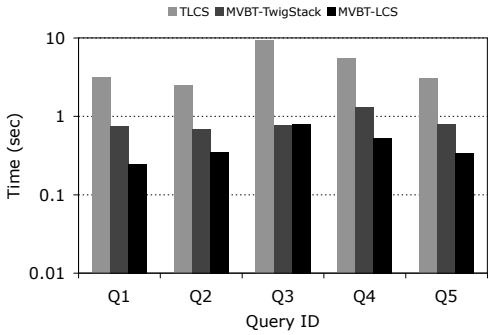

Figure 9: Unordered: Version Interval = 5

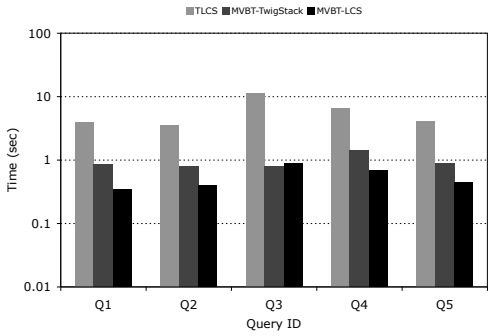

Figure 10: Unordered: Version Interval = 20

Nevertheless, the MVBT-LCS approach is still shown to be more efficient than all the other approaches with the exception of query Q_3, where MVBT-LCS is slightly slower than MVBT-Twigstack. In particular, Q_3 has many structural relationships, which create a large number of sequences needed to process unordered matches. In contrast, MBVT-Twigstack sequentially scans over the element labels effectively checking all possible combinations of a query in just one pass.

4.2.3 Average Runtime

We finally report the average query performance (over all queries) for each of the proposed algorithms, for ranges up to 50 versions. Figure 11, shows the average runtime of ordered matches. All algorithms show linearly increased runtime as the version interval increases (since more nodes need to be processed). Overall, the MVBT-LCS approach showed the most robust performance. The unordered queries showed similar behavior (full results reported in [3]).

5. CONCLUSIONS

As XML usage increases, so does the need to preserve and query the historical information of a document. Previous work on multiversion XML queries has either concentrated on modeling or simple path queries. In this paper, we examined how to process complex structural queries (twigs) over multiversion documents. We considered three approaches influenced by the current state-of-the-art techniques for static XML documents. This entailed identifying an appropriate labeling scheme that was both: (i) dynamic (i.e., it would not require tree re-labeling), and (ii) would integrate well with multiversion indexing. Our experimental results show

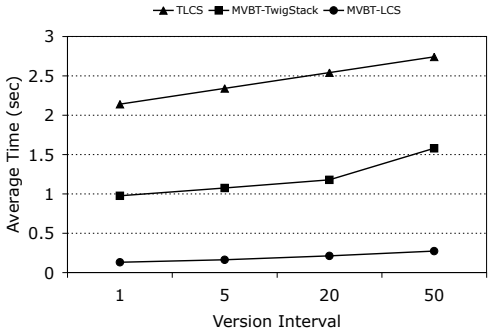

Figure 11: Avg. Time: Ordered Matching Queries

that the MVBT-LCS outperforms the other approaches. When unordered matches are needed and the twig query has high fan-out (thus resulting into many ordered twigs) the MVBT-Twigstack is a competitor. Overall, the proposed methods show low overhead for supporting multiple versions while using linear space on the number of updates.

6. REFERENCES

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

[2] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. Temporal databases: theory, design, and implementation. 1993.

[3] A. Woss. Query Processing Techniques Over Multiversion XML Documents. Masters thesis, UC Riverside, 2009.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[5] F. Wang, C. Zaniolo, X. Zhou, and H. J. Moon. Managing Multiversion Documents and Historical Databases: a Unified Solution Based on XML. In *Proc. of WebDB*, 2005.

[6] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *Proc. of ACM SIGMOD*, 2003.

[7] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. of ACM SIGMOD*, 2001.

[8] M. Moro, Z. Vagena, V.J. Tsotras. Tree-Pattern Queries on a Light-weight XML Processor. In *Proc. of VLDB*, 2005.

[9] Melvil Dewey. Dewey Decimal Classification and Relative Index 19th edition. Albany, NY, 1979.

[10] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. of ACM SIGMOD*, 2002.

[11] P. Buneman, S. Khanna, K. Tajima, and W-C. Tan. Archiving scientific data. In *Proc. of ACM SIGMOD*, 2002.

[12] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: insert-friendly XML node labels. In *Proc. of ACM SIGMOD*, 2004.

[13] P. Rao, and B. Moon. PRIX: Indexing And Querying XML Using Prufer Sequences. *Data Engineering, International Conference on*, 0, 2004.

[14] S. Tatikonda, S. Parthasarathy, and M. Goyder. LCS-TRIM: dynamic programming meets XML indexing and querying. In *Proc. of VLDB*, 2007.

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

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

[17] X. Wu, M. L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proc. of the 20th International Conference on Data Engineering*, 2004.

[18] XMark. The XML benchmark project. http://www.xml-benchmark.org.

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