UNIVERSITY OF CALIFORNIA
RIVERSIDE

Twig Queries Over Multiversion XML Documents

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Adam Woss

November 2008

Thesis Committee:

Dr. Committee Chair, Chairperson
Dr. Committee Member
Dr. Committee Member

Copyright by
Adam Woss
2008

The Thesis of Adam Woss is approved:

_____

_____

_____

Committee Chairperson

University of California, Riverside

ABSTRACT OF THE THESIS

Twig Queries Over Multiversion XML Documents

by

Adam Woss

Master of Science, Graduate Program in Computer Science
University of California, Riverside, November 2008
Dr. Committee Chair, Chairperson

Over the last few years XML has emerged as the standard for semi-structured data and exchange over the Internet. In recent years, a number of approaches have been proposed for storing the evolution of XML documents, thereby preserving useful temporal information. However, relatively little research has been done to adapt current XML querying techniques to take advantage of the temporal information being preserved. In this thesis, we discuss the design and development of a unified approach for temporal queries over multiversion XML documents. Specifically, we examine the aspects of managing the document over time and the algorithm used to carry out a query.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ability to query multiversion XML documents is a compelling problem that surfaces in many commercial and scientific applications. In general terms querying multiversion XML documents provides useful temporal information with regards to the version range specified in the query. However, current research lacks a unified approach to both managing multiversion XML documents and the method used for querying the XML document.

Over the last few years XML has emerged as the standard for semi-structured data and exchange over the Internet. Moreover, the vast majority of XML documents being disseminated undergo modifications (e.g additions, removals and updates) over time [10]. These modifications, in effect, create multiple versions of the XML document as time progresses, and in most cases the past versions are of historical importance. Consequently, devising an effective solution to storing multiversion XML documents has attracted a good deal of research interest over recent years [16][2][8][22].

In addition, relatively little research [25] has been done to adapt current XML querying techniques [13][21] to take advantage of the temporal information being maintained by multiversion XML documents. Instead, the two main techniques being employed for temporal XML queries either focus on converting XML documents to a relation tuples [4] or they use edit scripts to reconstruct the XML document for a given version [20]. Holistic processing methods on the XML data itself have been shown to outperform the former, while the latter fails to effectively handle range queries.

The remainder of this thesis will discuss the design and development of a novel approach for querying multiversion XML documents. Specifically, we examine the aspects of managing the document over time and the algorithm used to carry out a query.

Chapter 2

Background

An XML document is typically modeled as a graph or tree made of up nodes and values. Each node is assigned a label, which is a unique identifier that provides insight into the structural relationships. Figure 2.1 illustrates a simple XML document and its corresponding tree representation in Figure 2.2.

```
<Inventory>
    <Count>
        2
    </Count>
    <Item>
        <Name>Keyboard</Name>
        <Price>9.99</Price>
        <Description>Input Device</Description>
    </Item>
    <Item>
        <Name>Monitor</Name>
        <Price>175.00</Price>
        <Description>Displays Output<Description>
    </Item>
</Inventory>
```

Figure 2.1: Sample XML document data

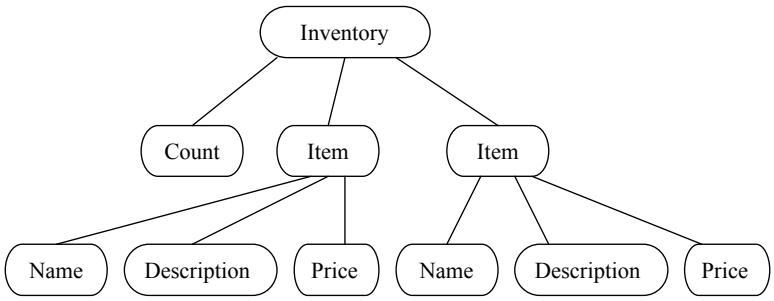

Figure 2.2: XML tree based off data in Figure 2.1

2.1 Labeling Schemes

A core operation of XML queries is being able to quickly determine parent/child and ancestor/descendant relations for a given set of tree nodes. In this section, we will examine several labeling schemes which have been proposed for XML documents.

Prior work focused on interval-based [19][13] labeling schemes, typically represented by the following tuple (DocId, LeftPos : RightPos, Level). Where (i) DocId is a unique identifier for a given document; (ii) LeftPos and RightPos represent the range of node labels contained by an elements descendants; lastly (iii) Level is the depth of the element. Given two tree elements E_1 and E_2, with intervals $(D_1, L_1 : R_1, L_1)$ 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 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. The exact range which should be used for the LeftPos and RightPos is unknown for each element, thus it becomes necessary to re-label the entire XML document. Figure 2.3 illustrates the interval scheme.

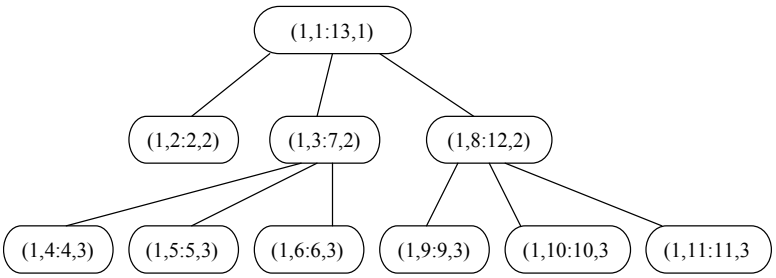

Figure 2.3: Interval-based labeling scheme

Another labeling scheme, known as prefix labeling [11][10][7], uses the prefix of a label to determine if an ancestor/descendant relationship exists. To establish a parent/child relationship it is required that prefix matches, as well as the length of each prefix must differ only by a single value. 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 aforementioned interval-based approach. Figure 2.4 demonstrates the prefix scheme.

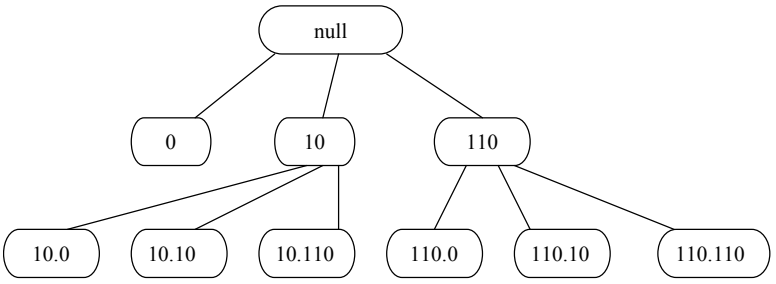

Figure 2.4: Prefix labeling scheme

The prime number labeling scheme [23] is a novel approach, maintaining tree order with-

out any re-labeling. Labels are the product of two values. The first, known as "parent-label" is inherited from the parent node. The second, called "self label" is assigned by the labeling scheme. This scheme exploits the property of prime numbers to ensure that each label can only be divided by its ancestor. To avoid re-labeling a new prime number is assigned to the "self-label" of newly inserted nodes. Although prime labeling supports ordered updates it has a couple disadvantages. First, the size of labels will become large as subsequent inserts require unique "self-labels". Second, re-calculation is both necessary and costly in order to determine structural relationships. Figure 2.5 displays the parent and self labels in a simple tree using prime labeling scheme.

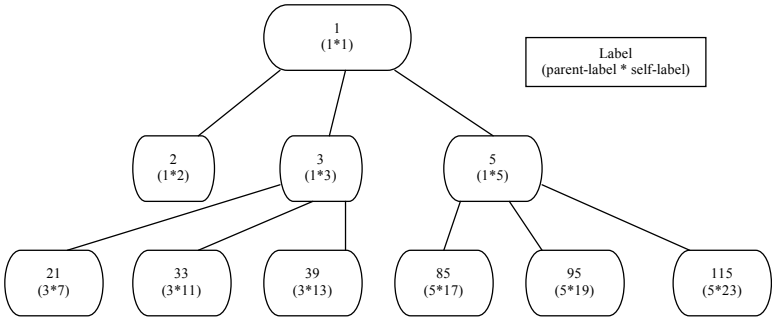

Figure 2.5: Prefix labeling scheme

Lastly, we review ORDPATH [15] which is a variant of the Dewey Decimal Classification [12]. Moreover, ORDPATH is a hierarchical scheme which improves upon similar prefix based methods discussed above. ORDPATH reserves even and negative values for subsequent inserts into the XML tree, which in turn allows for order to be maintained without re-labeling. The following is an example of an ORDPATH label "1.3.5.1", which can be compressed into a binary representation for simple substring matching of ancestry relationships. The method

in which ORDPATH inserts new nodes is called “careting in” and if this technique is applied frequently it could increase the length of labels significantly. However, O’Neil et al. [15] presented that excessive “careting in” is rare in practice. Figure 2.6 is an example of the ORDPATH labeling scheme.

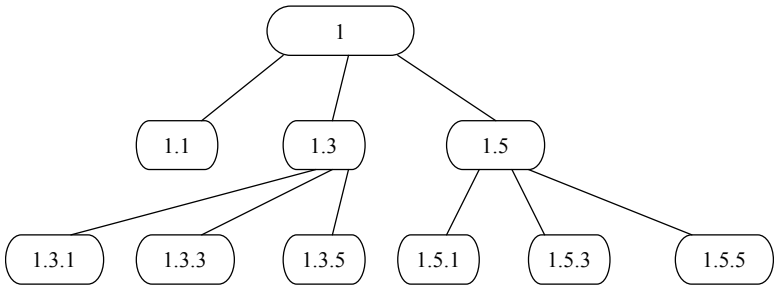

Figure 2.6: Prefix labeling scheme

2.2 Modeling Temporal XML Data

The topic of temporal databases has existed for some time [14][1], but as XML usage continues to grow so does the need to investigate more appropriate storage models relating to temporal XML data. In this section, we will examine the state of the art techniques being employed to manage temporal XML data. Furthermore, we exclude approaches that use temporal relational databases, as they tend to require complex extensions to both the database and SQL. Instead, we focus on XML conscious models designed specifically archiving changes undergone by XML documents.

The first approach we examine maintains a change-centric [2] representation of the XML data. The term change-centric means focusing on the changes themselves via the use of

deltas or edit scripts. This model relies on being able 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 model maintains all the changes made to a XML document, it suffers from a couple drawbacks. First, the XIDs are essentially a very primitive labeling scheme and as such do not provide any insight into ancestry relationships. Second, there is a significant cost incurred with having to reconstruct the document using deltas.

Another interesting approach relies on each node maintaing a set of timestamps [16] in order to archive the history of the document. This model can efficiently support the retrieval of any specific version as well as provide temporal history on elements. However, much like the delta approach, there lacks an efficient method to establish ancestry relationships between nodes. Moreover, there lacks any process to handle document order.

The referenced-based version model [22] aims to solve the limitations of edit-based approaches. The basic idea behind this model is 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. As shown in [22] the reference-based model has both improved storage and retrieval cost compared to more traditional approaches. However, there is concern that the management of the views could lead to increased overhead, as more and more versions are created.

The final approach utilizes an adaptation of a B-tree [5], which was originally developed to keep track of data over a set time. In fact, there has been various attempts at making the

B-tree persistent [3][6]. As proposed by Chen et.al in [22] the multiversion b-tree (MVBT) [3] can be used to manage temporal XML data. One convenient feature of the MVBT is that is already structured as a tree and unlike the referenced-based model, there is only a single temporal document from which versions can be retrieved. The MVBT model can easily be modified handle one of the XML labeling schemes discussed in the prior section.

2.3 XML Querying

Finding all occurrences of a query pattern is a fundamental operation for XML processing. Recent work has focused on holistic processing techniques, implying a global matching of the query pattern. Traditional methods have focused on decomposing the querying into multiple predicates and then merging the results; holistic matching has proven to be the superior technique. As a result, considerable research has focused on holistic techniques [13][21][17][9] for query pattern matching. Among the holistic techniques, are two differing approaches. The first relies on streams, which contain the element lists of each node in the query. These streams are then sequentially scanned to determine structural relationships. The second approach converts both the query and document into sequences and then performs some variation of subsequence matching.

Twigstack [13] is the original, in terms of stream based holistic algorithms, it consists of two phases: (i) streams of element lists are scanned, with stacks being utilized to store individual root-to-leaf solutions, and (ii) the partial solutions contained within the stacks are

merge-joined resulting in the answers to the query pattern. Furthermore, the merge-join step is optimal since it is guaranteed that any element pushed onto the stack must participate in the solution.

Recently LCS-TRIM [21] has presented new sequence based techniques, shown to be upwards of three orders of magnitude faster then preexisting sequence approaches PRIX [17] and ViST [9]. The speed up is attributed to the modification of the classic dynamic programming approach of longest common subsequence (LCS) to find all matches of a 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 potentially exponential. Nonetheless, LCS-TRIM is advantageous when ordered matching is necessary and Wang et al. [9] suggest twig queries are typically small in size and hence processing all possible variations of an unordered query may not be very expensive.

Chapter 3

Approach

Our approach focuses on extending modern holistic matching algorithms to operate on multiversion XML documents. The major challenge being that current holistic techniques were developed with a static XML data model in mind. We propose three modifications to the existing LCS-TRIM and Twigstack implementations. More precisely, in this section we examine the design and implementation or our: MVBT-Twigstack, MVBT-LCS, and TLCS algorithms.

3.1 MVBT-Twigstack

As the name suggests, this algorithm modifies the original Twigstack [13] indexing to use a MVBT. The advantages of using MVBTs as the element lists are the following: (i) efficient access to the elements of a given version number, (ii) coupled with ORDPATH labeling dynamic updates can be processed, and (iii) structurally the MVBT is similar to Twigstack's

XB-Tree. Figure 3.1 is an example of a MVBT on an element list.

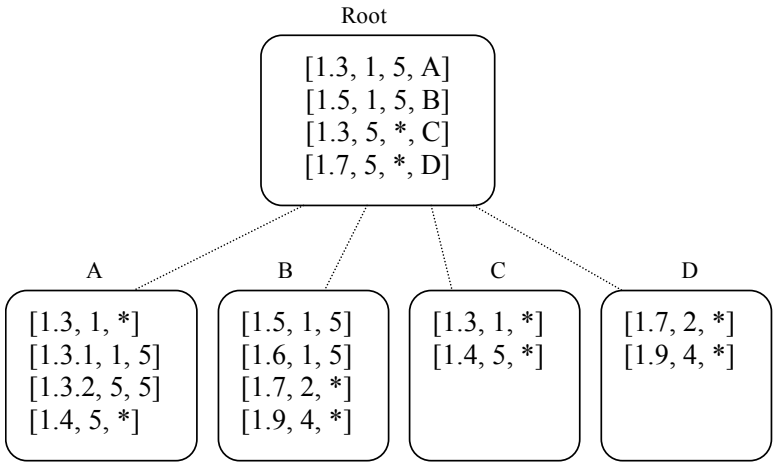

Figure 3.1: MVBT on element list

3.2 MVBT-LCS

This approach modifies LCS-TRIM [21] to incorporate a MVBT. Specifically, the XML document is itself represented as an MVBT, which allows for efficient access to only the needed versions of the document. Clearly, the MVBT will be of most use when the temporal range of a query is small, as it will allow for the pruning of non-relevant nodes without examining them first. After the MVBT has been traversed and the necessary nodes are isolated, the CPS and NPS can be constructed. The remainder of MVBT-LCS execution is exactly as described in Section 2.3.

3.3 TLCS

Temporal LCS (TLCS) relies on a timestamp XML data model, where each node in the XML tree maintains an interval representing the versions for which it was considered alive. Unlike the MVBT-LCS approach there is no way to efficiently construct the sequences required for LCS-TRIM. Moreover, we assume the sequences are computed offline, as such there is an extra pre-processing step incorporated into TLCS to eliminate irrelevant nodes. Despite the naive XML data model of TLCS, we hope this approach will provide meaningful insight into the performance of using different temporal XML data models.

Chapter 4

Results

In this section, we experimentally evaluate our modified algorithms: MVBT-Twigstack, MVBT-LCS and TLCS. As a baseline for our results we also include more traditional log-based and snapshot approaches in our experiments. In section 4.1, we present the set of queries used in out experiments. In section 4.1.1, we discuss the experimental settings. In section 4.2, we evaluate the storage cost of each approach. Finally, in section 4.3, we present the results of both ordered and unordered matching on a multiversion XML document using the queries presented in table 4.1.

4.1 Query Set

The queries in 4.1 are syntactically similar to XPath notation, however, the version range must be added to take advantage of the temporal information held within the multiversion XML document. We have attempted to create a set of queries which accurately demonstrates

Query 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 4.1: Twig queries for XMark data set

the advantages of both our methods and those proposed in literature.

4.2 Experimental Settings

All the experiments were run on a 2GHz Intel Core Duo with 4GB of main memory. We used the XMark[24] 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 [18] script was used to simulate new versions by applying a batch of inserts, removes and updates to the tree. In partiuclar, the data set generated has a size of 500MB and just over 6 million nodes.

4.3 Storage Cost

Figure 4.1 demonstrates the space usage as the XML document evolves over time. Clearly, the log-based approach has the minimal space usage, which is explained by not having the extra overhead associated with storing changes in a MVBT like structure. In contrast, the snapshot approach will quickly exceed main memory space as it tries to store every ver-

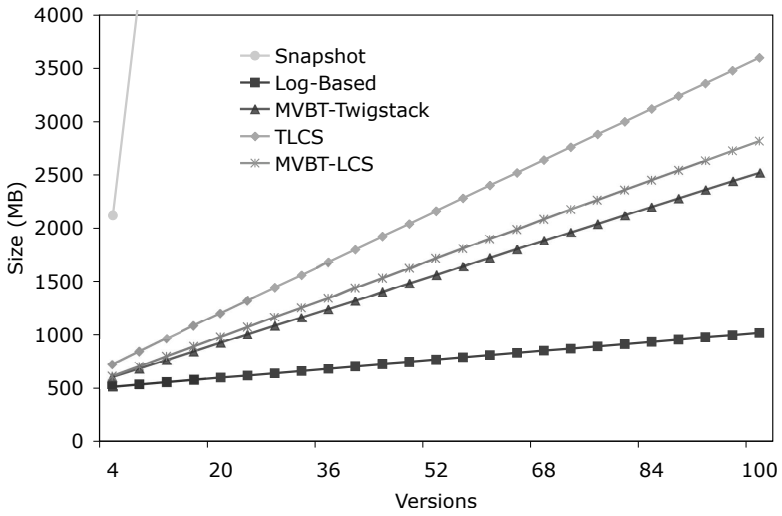

Figure 4.1: Storage Cost

sion. Although the snapshot approach may be able to retrieve a specific document version quickly, the space consumption clearly makes this approach impractical as number of document versions increase. TLCS, MVBT-LCS, and MVBT-Twigstack maintain space usage proportional to the changes made between versions. This behavior is ideal since there is efficient access to the needed nodes, without having excessive storage requirements.

4.4 Experimental Results

In this section, we analyze the performance of each algorithm using the queries given in Table 4.1. We considered four different temporal ranges for each query: 1, 5, 20 and 50. For example, a temporal range of 5 corresponds to a query only spanning 5 versions of the document, likewise, a temporal range of 1 targets only a single version. In section 4.3.1, we show the results of ordered matching. In section 4.3.2, we present the results of unordered

matching. And in section 4.3.3, we examine the average runtimes of all algorithms.

4.4.1 Ordered Matches

Figures 4.2-4.5 illustrate the execution time of $Q_1 - Q_5$ for each of the temporal ranges listed above, respectively. The Snapshot approach clearly performs well when the temporal range is set to 1, since it has every version individually stored on disk. However, as the temporal range increases the performance quickly degrades, which is due to the cost of trying to merge results from each document version. While the Log-Based method provided the minimal amount of storage space, its query runtimes are too expensive regardless of temporal range. This is attributed to the overhead incurred when having recreate the document for a given versions based off the data stored in the logs.

Both TLCS and MVBT-Twigstack perform adequately well, as demonstrated in Figures 4.2-4.5, with MVBT-Twigstack just edging out TLCS in terms of execution speed. We observe that as the temporal range increases it does not correlate to increased query times in the TLCS approach. This is explained by the fact that TLCS has the entire subsequence that must be parsed and all elements will be examined no matter what the temporal range is. In contrast, MVBT-Twigstack and MVBT-LCS only examine the needed nodes, which means as the temporal range increases the number of nodes accessed by either MVBT-Twigstack or MVBT-LCS must also increase. Overall, the performance of MVBT-LCS is clearly better then any of the other approaches. In particular, the subsequence matching of LCS-Trim proves to be a more efficient approach compared to that of Twigstack.

Lastly, we examine the effect of they queries $Q_1 - Q_5$ on the runtime of each approach. Both Q_1 and Q_5 are basic twig queries with low selectivity. As expected, the runtimes with regards Q_1 and Q_5 are among the fastest for all experiments. In contrast, Q_3 and Q_4 contain a higher fan-out and depth, which yield slower runtimes.

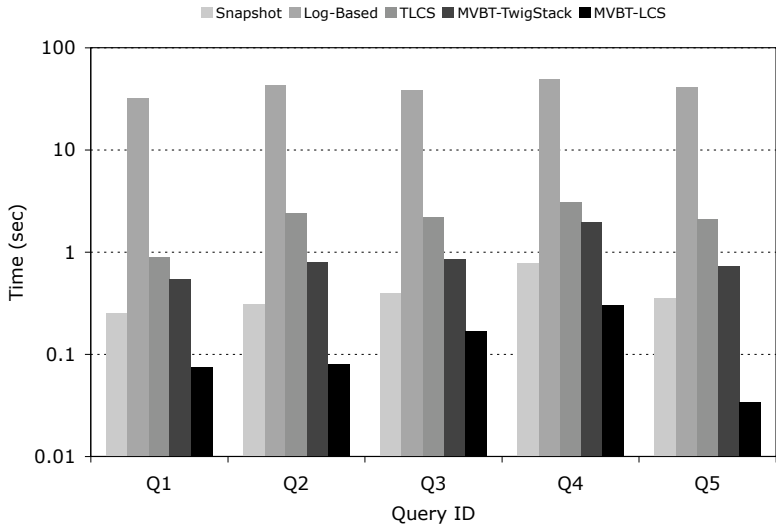

Figure 4.2: Execution Time of Ordered Results on Version 1

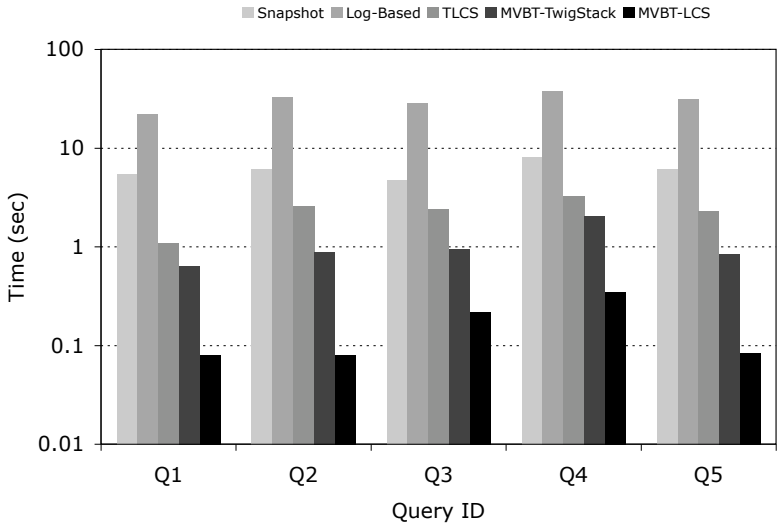

Figure 4.3: Execution Time of Ordered Results on Versions 11-15

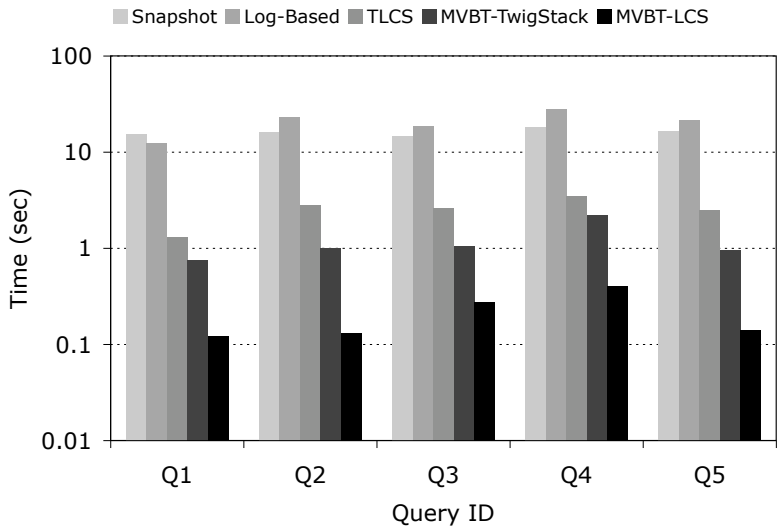

Figure 4.4: Execution Time of Ordered Results on Versions 11-30

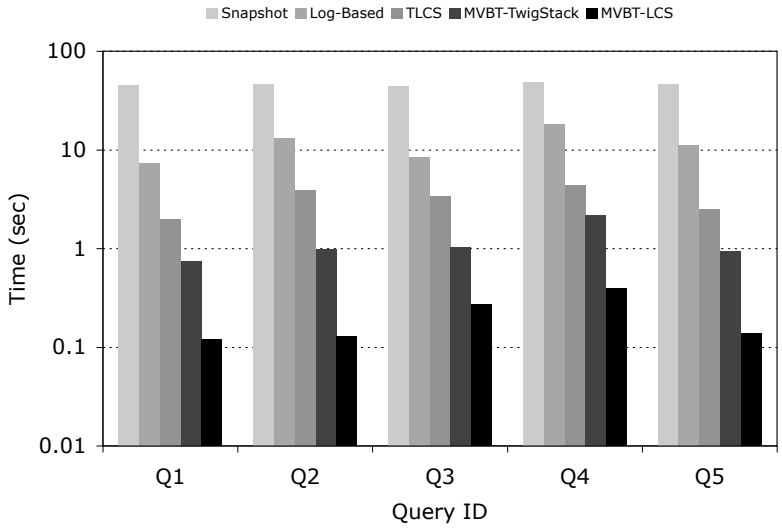

Figure 4.5: Execution Time of Ordered Results on Versions 1-50

4.4.2 Unordered Matches

We will now compare the results when unordered matches are desired. Figures 4.6-4.9 illustrate the execution time of $Q_1 - Q_5$ employing the same experimental settings used for ordered matches discussed above. Clearly, there are common characteristics between both ordered matches and unordered matches in terms of overall performance of each approach. For instance, MVBT-LCS proves to be more efficient then all the other approaches just as it was for ordered matches. The only exception to this is Q_3 where MVBT-LCS is slightly slower then MVBT-Twigstack. This is explained by the fact that Q_3 has many structural relationships, which create an exponential number of sequences needed to process unordered matches. Recall; the original LCS-Trim is based on tree traversals resulting in order among sibling nodes, meaning all configurations of query must be processed for unordered matching. In contrast, the performance of MVBT-Twigstack is maximized when unordered matches are of concern. This is attributed to the sequential scan over element labels performed by MBVT-Twigstack, which effectively checks all possible combinations of a query in just one pass.

4.4.3 Average Runtime

We now compare the overall performance for each of the proposed algorithms. In Figure 4.10, we show how the average runtime of ordered matches with respect to the temporal range. The Snapshot approach is clearly affected most when queries extend over large version intervals. For example, the average runtime of the Snapshot approach increases nearly two

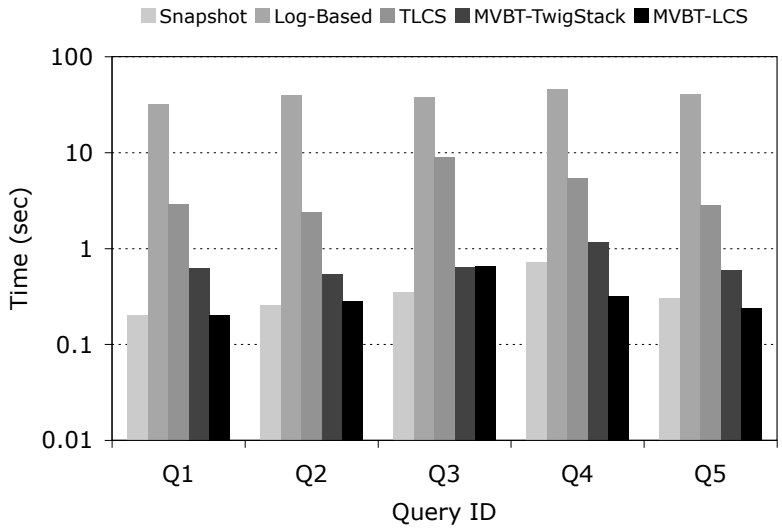

Figure 4.6: Execution Time of Unordered Results on Version 1

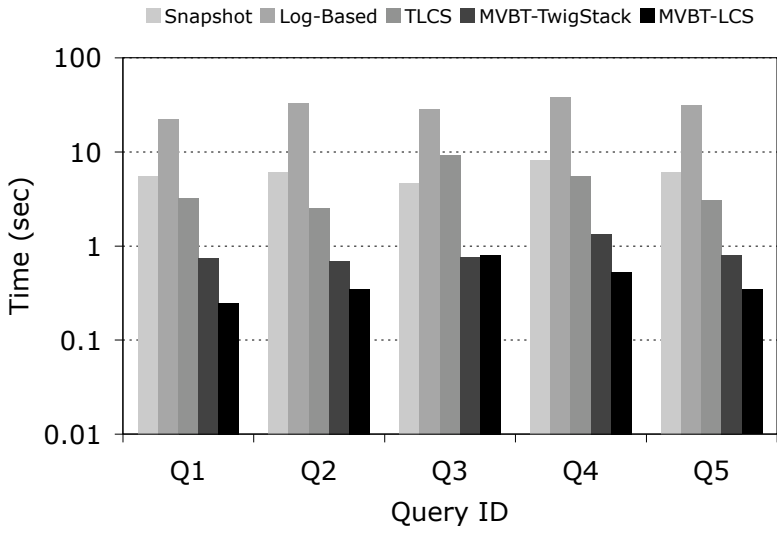

Figure 4.7: Execution Time of Unordered Results on Versions 11-15

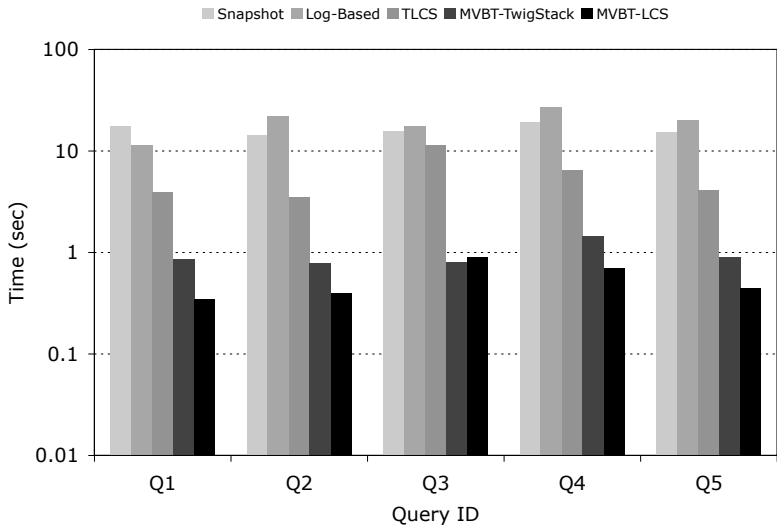

Figure 4.8: Execution Time of Unordered Results on Versions 11-30

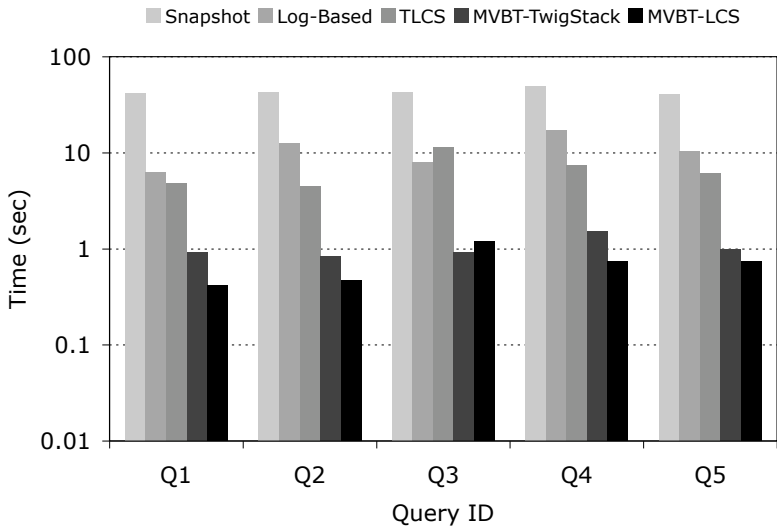

Figure 4.9: Execution Time of Unordered Results on Versions 1-50

orders of magnitude in Figure 4.10. This decrease in performance is due to the heavy cost incurred by having to query each document version separately and then merge the results. The largest performance gap between MVBT-LCS and MVBT-Twigstack is evident in Figure 4.10 when we compare average runtimes for ordered matching. However, that gap is quickly closed when looking at Figure 4.11 showing the average runtime for unordered matching.

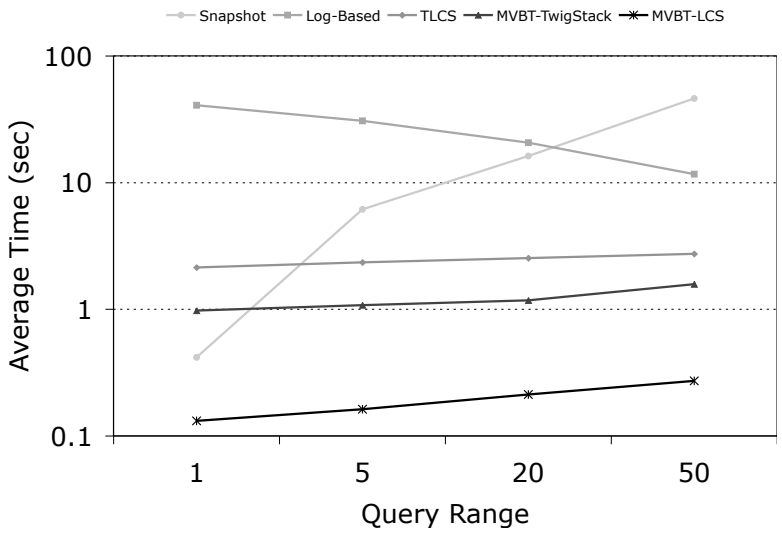

Figure 4.10: Average Time of Ordered Matching

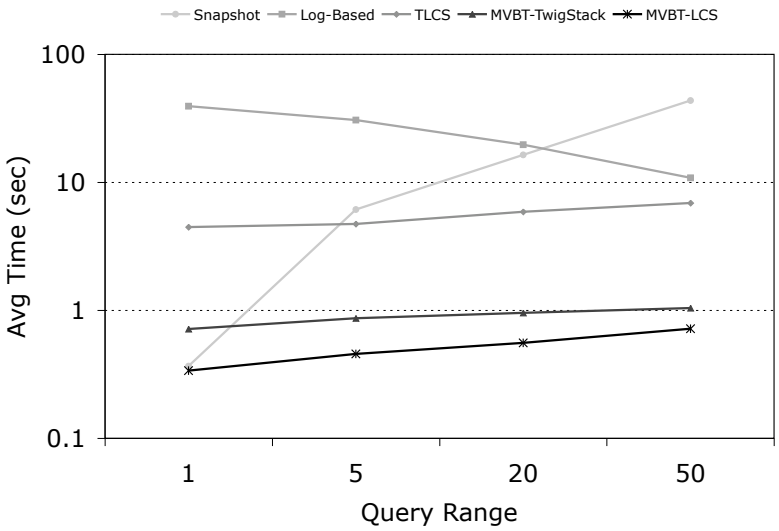

Figure 4.11: Average Time of Unordered Matching

Chapter 5

Conclusion

As XML usage continues to increase, so does the need to preserve the historical information of the document. A significant amount of work has focused on the problem of preservation and several suitable solutions have been proposed for modeling multiversion XML documents. Nevertheless, the problem of querying these multiversion XML documents has gone relatively untouched.

In this thesis, we have introduced the design of development of several techniques which address querying multiversion XML documents. More specifically, we expanded and modified the current state-of-the-art techniques, for querying static XML documents, to work with multiversion XML documents. This entailed finding a labeling scheme that was both: (i) dynamic in the sense that it would not require tree re-labeling, and (ii) would integrate into the multiversion model we choose to use. Our experimental results show that MVBT-LCS outperforms the other approaches most of the time. The only domain where MVBT-LCS is not

ideal is one in which unordered matches are needed and the twig queries have high fan-out.

Bibliography

[1] Abdullah U. Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. Temporal databases: theory, design, and implementation. Benjamin-Cummings Publishing Co., 1993.

[2] Amelie Marian, Serge Abiteboul, Gregory Cobena, and Laurent Mignet. Change-Centric Management of Versions in an XML Warehouse. In *Proceedings of the 27th international conference on very large data bases*, pages 581–590, 2001.

[3] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[4] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 425–436, 2001.

[5] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[6] David Lomet, and Betty Salzberg. Access Methods for Multiversion Data. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 315–324, New York, NY, USA, 1989. ACM.

[7] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic XML trees. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 271–281, 2002.

[8] Fusheng Wang, Carlo Zaniolo, Xin Zhou, and Hyun J. Moon. Managing Multiversion Documents and Hisorical Databases: a Unified Solution Based on XML. In *Proceedings of WebDB*, 2005.

[9] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 110–121, New York, NY, USA, 2003. ACM.

[10] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 413–424, 2001.

[11] Maggie Duong, and Yanchun Zhang. LSDX: A New Labelling Scheme for Dynamically Updating XML Data. In *Proceedings of the 16th Australasian database conference*, pages 185–193, 2005.

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

[13] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, 2002.

[14] Opher Etzion, Sushil Jajodia, Suryanarayana Sripada. Temporal Databases: Research and Practice. Springer, 1998.

[15] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908, 2004.

[16] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 1–12, 2002.

[17] Praveen Rao, and Bongki Moon. PRIX: Indexing And Querying XML Using Prufer Sequences. *Data Engineering, International Conference on*, 0:288, 2004.

[18] Python programming language. http://www.python.org.

[19] Quanzhong Li, and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th international conference on Very large data bases*, 2001.

[20] Raymond K. Wong, and Nicole Lam. Managing and querying multi-version XML data with update logging. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 74–81, 2002.

[21] Shirish Tatikonda, Srinivasan Parthasarathy, and Matthew Goyder. LCS-TRIM: dynamic programming meets XML indexing and querying. In *Proceedings of the 33rd international conference on Very large data bases*, pages 63–74, 2007.

[22] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Management of Multiversion Documents by Object Referencing. In *Proceedings of the 27th international conference on Very large data bases*, pages 291–300, 2001.

[23] Xiaodong Wu, Mong Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proceedings of the 20th International Conference on Data Engineering*, 2004.

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

[25] Zografoula Vagena, and Vassilis J. Tsotras. Path-expression Queries over Multiversion XML Documents. In *Proceedings of the international workshop on the Web and Databases*, pages 49–54, 2003.