UNIVERSITY OF CALIFORNIA
RIVERSIDE

Structured Queries Over XML Documents with Branched Versioning

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

Master of Science

in

Computer Science

by

Wanxing Xu

June 2009

Thesis Committee:

Professor Vassilis J. Tsotras, Chairperson
Professor Mart Molle
Professor Eamonn Keogh

The Thesis of Wanxing Xu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I thank my committee, especially the chairperson, my advisor Professor Vassilis J. Tsotras, without whose help, I would not have been here.

To my parents.

ABSTRACT OF THE THESIS

Structured Queries Over XML Documents with Branched Versioning

by

Wanxing Xu

Master of Science, Graduate Program in Computer Science
University of California, Riverside, June 2009
Professor Vassilis J. Tsotras, Chairperson

Over the last few years, there is an increasing use of XML documents as the standard for tree-structured data. There are several research done for querying and indexing an XML database, especially finding all occurrences of a twig pattern. However, those research focus on the static XML documents, but not data with multiple versions. On the other hand, there are also a number of approaches have been proposed to support temporal data with branched time evolution, called branched versioning database. Different from conventional temporal databases, in a branched versioning database, both historic versions and current versions are allowed to be updated. Relatively little research has been done across the two areas. In this thesis, we discuss the design and development of some unified approaches for temporal queries over XML documents with branched versioning. Specifically, we examine the storage and querying of the branched versioning XML documents and the algorithm used to carry out a twig pattern query.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last few years, there is an increasing use of XML documents as the standard for tree-structured data over the Internet. Moreover, the vast majority of XML documents being disseminated undergo modifications (for example, insertions, deletions and updates) over time. In most cases, the past versions are of historic importance so that we need to manage every version of the XML document with the effects of the modifications as the time progresses. As a result, many research interest has been raised to devise an effective solution to storing multiversion XML documents.

Conventional temporal databases assume a single line of time evolution, called data with linear versioning, which allows only the latest version to be modified, i.e. a new version is created by modify only the latest version at that time. However, under some circumstances, both the latest version and a historic version maybe need some modification. A modification on a historic version may give a branch on the time evolution relationship, because two or more versions are evolved from the same version. Version trees are used

to describe the evolution relationship between different versions. These kind of data with branched versioning cannot be supported by conventional databases of linear time evolution.

Many applications require the ability to do the structured queries over XML documents with branched versioning. In general terms, the version is specified in the query with the twig pattern as well, according to which, the querying results will provide the useful temporal information. Though there are many research done for twig pattern matching for single version (or static) XML documents [6] [11] [15] [13], and several approaches to manage data with branched versioning [8] [9], there is a lack to combine these two parts together, i.e. to support the twig pattern matching over XML documents with branched versioning.

The remainder of this thesis will discuss the design and development of two novel approaches for querying XML documents with branched versioning. Specifically, we examine the storage and querying of the branched versioning XML documents and the algorithm used to carry out a twig pattern matching at a specific time during the evolution.

Chapter 2

Background

Extensible Markup Language (XML) is a markup language which allows users to define new collections of tags that can be used to structure any type of data or document the user wishes to transmit. Tags, also called elements, are the primary building blocks of an XML document. Figure 2.1 illustrates a simple XML document.

```
<SHOPPINGLIST>
        <DATE>2009-5-14</DATE>
        <COUNT>2</COUNT>
        <ITEM>
                <NAME>ballpens</NAME>
                <AMOUNT>1 dozen</AMOUNT>
        </ITEM>
        <ITEM>
                <NAME>notepad</NAME>
                <AMOUNT>2 or 3</AMOUNT>
        </ITEM>
</SHOPPINGLIST>
```

Figure 2.1: A simple XML document

An XML document is typically modeled as a tree made up of nodes and values. Each node represents an element in the XML document. The parent-child relationships of

the nodes in the tree also represent the structural relationships of the elements in the XML document. Figure 2.2 illustrate the tree representation of the XML document in Figure 2.1.

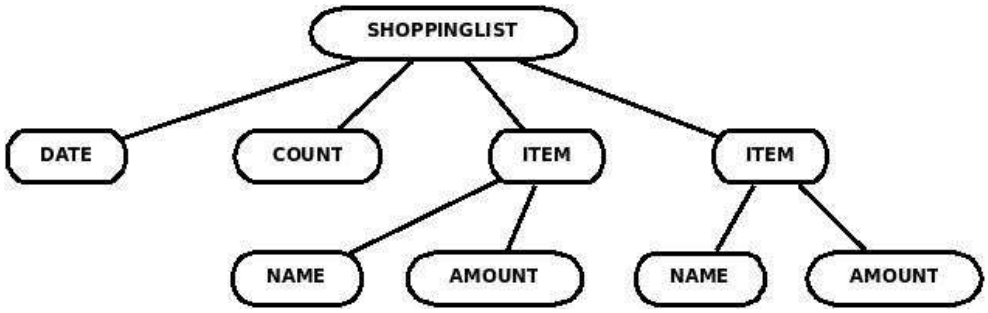


Figure 2.2: Tree representation of the XML document in Figure 2.1

2.1 Labeling Schemes

To quickly determine the parent-child or ancestor-descendant relationships for a given set of tree nodes is required as a core operation of XML query matching. With only the original XML document, this task cannot be easily done. A good labeling scheme can make this operation more efficient. Another aspect that we need to concern is that we are dealing with the temporal data, which intent to be changed a lot. A good labeling scheme in this situation should has little affect to the unrelated part by modifications of the document. In this section, we will briefly examine several labeling schemes based on the two requirements mentioned above.

2.1.1 Interval-based Labeling Schemes

An interval-based labeling scheme is commonly used in several prior work [3] [6], which represents each node by a 3-tuple (DocID, StartPos : EndPos, Level), where (i) DocID is a unique identifier for a given document, (ii) StartPos and EndPos is the position of the start and the end of the element (the position can be generated by counting word numbers from the beginning of the document) and (iii) Level is the nesting depth of the element in the document. Figure 2.3 illustrates the same XML document from Figure 2.1 using this labeling scheme.

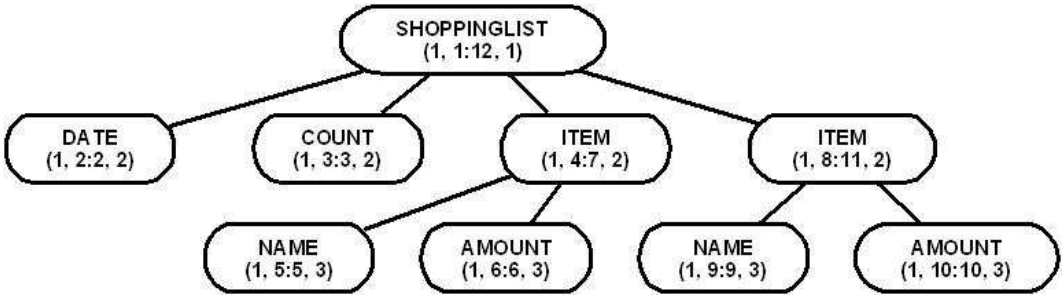


Figure 2.3: Interval based labeling scheme of the XML document in Figure 2.1

With the 3-tuple representation, we can easily determine the parent-child and ancestor-descendant relationships. Given two element $N_1(D_1, S_1 : E_1, L_1)$ and $N_2(D_2, S_2 : E_2, L_2)$. N_1 is an ancestor of N_2 if and only if both the following two conditions hold: (i) the two elements belong to the same document, i.e. $D_1 = D_2$; (ii) the interval of N_1 includes the interval of N_2, i.e. $S_1 < S_2 < E_2 < E_1$. In addition, that N_1 is the parent of N_2 requires the same two conditions hold and also an additional condition that N_1 is only one level above N_2, i.e. $L_1 + 1 = L_2$.

Though this interval-based labeling scheme works very well for the requirement of determine the parent-child and ancestor-descendant relationships fast, it doesn't do well if there is an insertion or deletion in the XML document. Consider now we want to add the information of the writer of the shopping list by inserting several new elements in the middle of the XML document. The original XML document is called Version 1 and the new XML document is called Version 2, which is illustrated in Figure 2.4. Figure 2.5 shows the tree representation, where the modification inserts a subtree to the original tree with a dark background of the nodes.

```
<SHOPPINGLIST>
        <DATE>2009-5-14</DATE>
        <COUNT>2</COUNT>
        <WRITER>
                <NAME>John Doe</NAME>
        </WRITER>
        <ITEM>
                <NAME>ballpens</NAME>
                <AMOUNT>1 dozen</AMOUNT>
        </ITEM>
        <ITEM>
                <NAME>notepad</NAME>
                <AMOUNT>2 or 3</AMOUNT>
        </ITEM>
</SHOPPINGLIST>
```

Figure 2.4: The sample XML document data of Version 2

Because the interval-based labeling scheme is using the position generated by counting word numbers from the beginning of the document, an insertion or deletion will affect all the ancestor elements (for the end positions) and all the elements appears after where the change occurs (for both the start and end positions). Figure 2.6 shows the interval-based labels of the XML document of Version 2. A big part of the tree is affected

Figure 2.5: Tree representation of the sample XML document of Version 2

by the modification, so we can see that this interval-based label scheme doesn't work well for the XML documents of multiple versions.

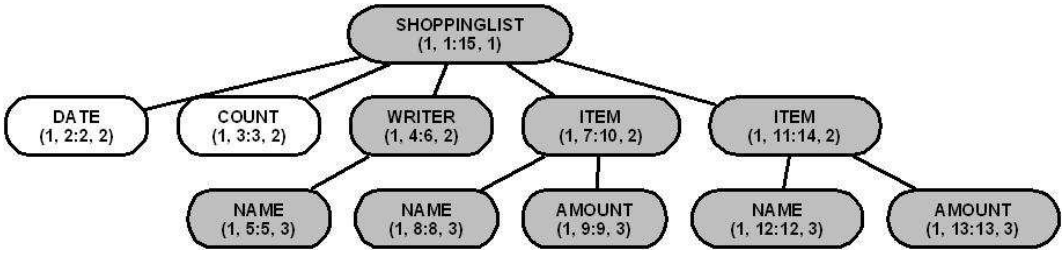


Figure 2.6: The interval-based label schemes of the sample XML document of Version 2

2.1.2 ORDPATH

Now, we review another labeling scheme called ORDPATH [10], which meets both of the two requirements: (i) fast determination of parent-child and ancestor-descendant relationships, and (ii) little affect by modifications.

ORDPATH is a hierarchical scheme, which encodes the parent-child relationship by extending the parent's ORDPATH label with a component for the child. Figure 2.7 illustrates the ORDPATH labels for the original sample XML document in Figure 2.1. We

can easily check the ancestor-descendant relationships between two elements by checking whether the ORDPATH label of one element is a prefix of that of the other. And for parent-child relationships, just check one more condition for the difference of their levels (the level is known by counting the odd numbers in the label, we will discuss this more later). For example, the element "1.5.3" is a child of "1.5" and a descendant of "1". And the number itself keeps the children of the same element to be ordered (smaller numbered child appears before the bigger ones).

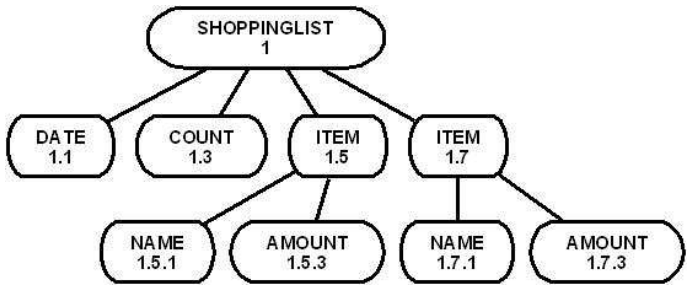


Figure 2.7: The ORDPATH label scheme of the sample XML document

In the initial load of ORDPATH labeling, only positive odd integers are used, while the negative numbers and even numbers are reserved for later insertions. If the insertion is to the right of all its siblings, we can number it by adding two to the last ordinal of the last sibling (the next odd number). And if the insertion is to the left of all the siblings, we subtract two from the last ordinal of the leftmost sibling (the previous odd number, must be negative). These two cases are still using odd numbers, bigger positive number on the right while smaller negatives on the left.

If an insertion is between two sibling, the even number is used as a "caret", then following this with a new odd component, which is as the same as the initial load. We

can using the example of the Version 2 of the sample XML to show how ORDPATH works for insertions. Figure 2.8 shows the ORDPATH labels of the Version 2. The new element of "WRITER" is inserted between its two sibling elements "COUNT" which is labeled as "1.3" and "ITEM" as "1.5". Now we will use 4 (which is the even number between 3 and 5) as a "caret" and another odd number 1 (the first positive odd number) together as the component for the new element. So, in total, "1.4.1" as the whole label of the new element. Though "1.4.1" has three components, it is still a second level element (the same level as "1.3" and "1.7"), because only odd numbers are counted as the level but not even numbers. Even numbers are only symbols of carets and can be used to check the order of the siblings, but don't mean one more level. If we want to insert another element between "WRITER" and "ITEM", we can still use "1.4.3" as it is to the right of "1.4.1", who is the only child of the pseudo node "1.4".

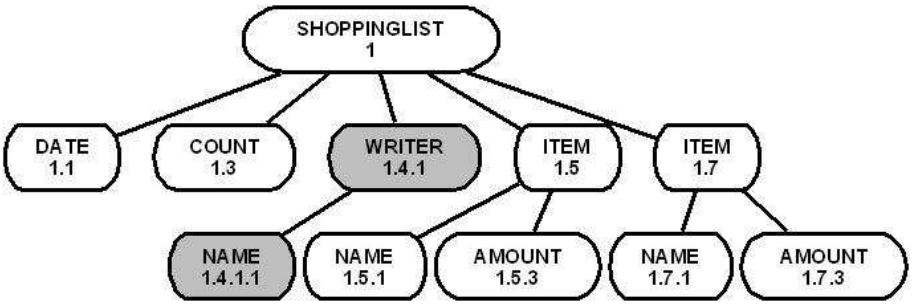


Figure 2.8: The ORDPATH label scheme of the sample XML document of Version 2

2.2 XML Querying

In the recent years, a lot of research interests have been paid on the XML query matching, which requires to find all occurrences of the query pattern in the XML documents. As the XML documents are in the tree structures, the query patterns can also be in the tree structures. At the early time of the research in this area, only one pair of ancestor-descendant or parent-child can be found in the XML documents at one time, then the huge intermediate results for each pair are joined together to get the final results for the whole query pattern [3]. This method requires multiple passes to match one twig pattern because it has to break a twig pattern into several nodes pairs, and a lot space is needed to hold the huge intermediate results to later be joined together. After that, PathStack [6] can match one whole path in the query pattern at one time with the help of stacks. It works better then the former one, though the query still has to be first split into several paths, processed separately and joins need to be done among the big intermediate results.

Recent work has focused on holistic processing techniques, implying a global matching of the whole query pattern without cut the twig into pieces. There are two general catalogs of the methods doing the holistic query processing: the first catalog includes the algorithms based on streams of the elements, where a stream is a list of all the occurrences of one particular element and these streams are sequentially scanned to determine the structural relationships; the other algorithms first convert the tree-structured XML documents and twig patterns into linear sequences, and then use subsequence matching and structural refinements, instead of the tree pattern matching. We call the algorithms in the first catalog to be stream-based algorithms and the second to be sequence-based algorithms.

TwigStack [6] is one of the well-known stream-based XML holistic query process algorithms. It improved the PathStack which splits the twig query into several path queries, processes only one path at a time and merges the results at last. In TwigStack, the twig query is processed as a whole: streams of the element lists are scanned sequentially for every element in the query, with stacks being utilized to store individual root-to-leaf solutions (as same as PathStack does). The stacks hold partial solutions for the query unless each individual elements gets its own solutions, merging which together gives solutions to the whole query. The algorithm guarantees that each element pushed into the stack must participate in at least one solution, so it does not produce any intermediate results that is not in a final solution.

Recently, there are several algorithms using the sequence-based techniques to do the twig pattern matching, which belong to the second catalog mentioned above. Generally, those approaches convert the tree-structured XML documents and twig pattern into linear sequences and then use subsequence matching together with structural refinements to do the query process. The structural information should be kept in the linear sequences so that the structural refinement can be executed to avoid false alarms in the result set.

ViST [15] converts the tree-structured data into only one sequence. Each node is represented as a 2-tuple (label, prefix), where the prefix indicates the labels on the path from the root to this node. The nodes appear in the sequence in the pre-order in the tree. First, ViST will check whether the sequence converted from the twig pattern is a subsequence of that of the documents, where it uses suffix-trees. And then, the structural relationship of the twig and documents are checked as the structural refinements. When

both the subsequence relationship (called S-Ancestorship) and the structural relationship in the original XML documents (called D-Ancestorship) hold, it is considered to be a solution to the query.

PRIX [11] and LCS-TRIM [13] are also sequence-based algorithms where some variations of *Prüfer* sequence are used. These two algorithms convert one XML document into two linear sequences: numbered *Prüfer* sequence and labeled *Prüfer* sequence. They used the post-order to unique numbering the tree nodes. PRIX uses several phases to do the query process: (i) subsequence matching for the labeled *Prüfer* sequence, (ii) refinement by connectedness, (iii)refinement by twig structure, (iv) refinement by matching leaf nodes and then (v) processing wildcards if any. PRIX needs to keep the intermediate results for each phase which cost much time and space. LCS-TRIM uses only two phases: subsequence matching and structure matching, which can be put together and false positive matches can be pruned early on the go. It uses the dynamic programming approach of the Longest Common Subsequence (LCS) to find all subsequences of a query. It is shown to be upwards of three orders of magnitude faster than PRIX and ViST.

2.3 Management for Data of Multiple Versions

Many applications tent to have variations for the data and the past data are also of historic importance that needs to be kept. Therefore, the management (storage and querying) of these kind of temporal data is acquired and attracts some research interests. There are two kinds of naïve approaches: log-based approach and snapshot approach. Log-based approach only store the log of all the modifications to the data. When querying for

the data at a specific time, it will reconstruct the data by redoing the modifications in the log from the very beginning to the querying time, which is costly in time. The Snapshot approach simply store every version of all the data. The parts which are not modified will have many copies stored for different versions, which cost a lot of space. A more efficient approach on both time and space is preferred to these two naïve approaches.

2.3.1 Multiversion B-Tree

Multiversion B-Tree (MVBT) [4] is an efficient approach to store and query linear versioning data. The linear versioning data allows modification only to the latest version. As the name suggests, the Multiversion B-Tree uses an indexing tree similar to B-Tree [7], but the difference is that in MVBT, to compare the entries in one node to choose which subtree to follow, both key and time values are used. Each data entry has a key value, a start time (when it is inserted), an end time (when it is deleted, or '*' to indicate it is not yet deleted or still "alive") and the information of the data entry if necessary. The index entry has a key value, a start time and an end time (of all the entries in the subtree rooted from the node it pointing to) and the pointer to the node of the next level. As B-Trees, MVBT doesn't allow underflow or overflow of physical entries in one node (which is called weak version conditions in MVBT). It also has an additional strong version conditions to constraint the number of alive entries inside one node. Merging or Splitting will took place if any of the strong or weak version conditions doesn't meet. The weak version condition ensures the utilization of the space and the strong version condition puts the entries close in time to be close in storage which ensures a short querying time.

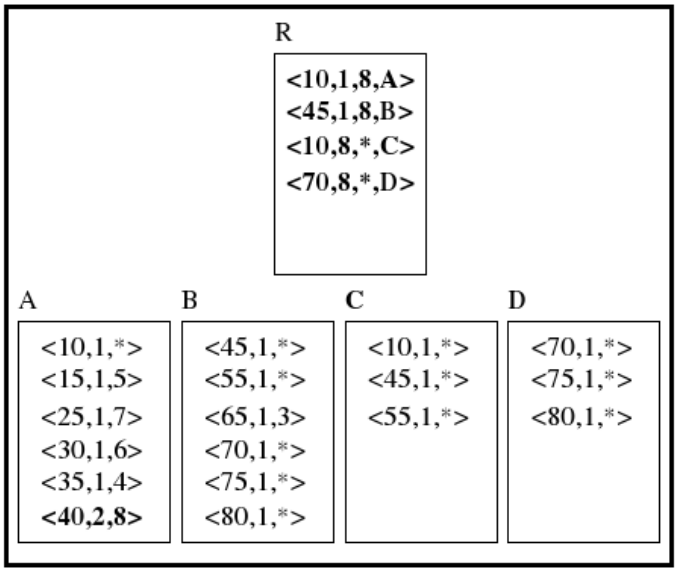


Figure 2.9: A sample Multiversion B-Tree

Figure 2.9 illustrate an example of MVBT. We see the root index page R contains four entries which direct search into different data pages. For example, a search for a key between 10 and 45 and at time between 1 and 8 will go to the data page A. Because there is only one entry $\langle 10, 1, * \rangle$ in the page A that is alive, the entry is copied to a new node and page A is considered died at time 8. This entry $\langle 10, 1, * \rangle$ is merged with the alive entries from the page B then further the overflow page is split to be two new pages C and D and page B is also marked died at time 8. Though there are two copies of the alive entries, the search is faster because the already died data will not be processed.

2.3.2 BT-Tree and BTR-Tree

The MVBT works fine for the linear time evolution temporal data, where a modification can only happen to the latest version, though it cannot support data with branched

versioning. Under some circumstances, modifications occur on not only the latest version, but any historic version as well. The modification of a historical version will give a branch on the evolution relationships between versions. We call these kind of data the data with branched versioning to distinct from the linear versioning. We will use the following example to illustrate this kind of data.

For example, Alice is making a travel plan for her vocation and she is still changing the details where each modification gives a new version of her plan. For her, every new version is created after some modification of the latest version. Bob wants to travel to the same place, too. At time 8, Bob starts using the Alice's plan at Version 5 and makes his own modification according to his interests. So this change is on a historical version but not the latest version (which should be Alice's plan at time 7). And meanwhile, Alice is still making her changes. At time 10, a third person David starts using Alice's plan at Version 10 and also makes his own changes. The above are data for the travel plans with branched versioning. Each person holds a different version of the data: we call Alice's plan to be Version 1, Bob's Version 2 and David's Version 3. And for each person has a linear time evolution of the plan, we mark them with the time they are created: we call Alice's plan created at time 1 as Version 1.1, Bob's plan created at time 12 as Version 2.12 and so on. So each Version is represent as a 2-tuple with the version id and time, e.g. (1, 1) and (2, 12). In Figure 2.10, a version tree gives the relationship among those different versions.

Rather than this graphic view for the relationship of the versions, a version table is used which records the information of evolutions of each version. Table 2.1 shows the version table represent the same version relationship as in the version tree in Figure 2.10.

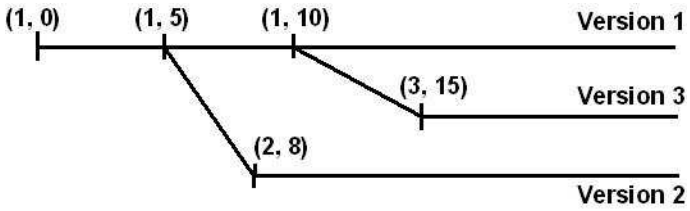


Figure 2.10: The version tree for the travel plan example

In our example, Bob's first version of the travel plan was created at time 8, which is the version (2, 8), inherit from Alice's plan at time 5 which is (1, 5). So we record: a new Version 2, starts at time 8, shares the same data with Version 1 at time 8. An entry (2, 1, 8, 5) is inserted into the version table correspondingly as shown in the second line. To check the relationship between two versions, we can easily tell that a version with earlier time is an ancestor of that of the same version but a later time. If the two version is not with the same version id, we can check with this version table, to know their relationships.

Version id	Ancestor version id	Start time	Share time
1	1	0	0
2	1	8	5
3	1	15	10

Table 2.1: The version table for the version tree in Figure 2.10

To effectively store and query is a main requirement for the data management. The two naïve approaches mentioned above, log-based approach and snapshot approach, both work for data with branched versioning, though the same problem, either time-consuming or space-consuming, is still existing. MVBT cannot be easily adopted for this problem because here, the same data may be shared across several branches, each of which may be

modified separately. MVBT cannot tell which version the modification is on or it has to have multiple copies of the data to tell so, which may be inefficient. Moreover, Linan Jiang, Betty Salzberg, et al. give some good solution in [8] [9] [12].

The BT-Tree [8] proposed a paginated access method for data with branched versioning (or called "branched-and-temporal data" there), which was later improved by the BTR-Tree [9] and better explained by a framework [12]. The main idea is to create a tree as an index to fast lead the search to the page which contains the data. The data are put into pages, and when a page is full, a version split or a version-and-key split will occur according to different situation. A version split at version v will separate the data of a descendant version of v from those who are not. Only alive records will be copied and again it may cause another overflow. Consequently, an additional key split will occur, which will make this split to be version-and-key split. The key split is a B-tree like split: a key split at key k will separate values less than k from those greater than or equal to k.

The indexes of the data are also put into pages, which are called indexing pages. The indexing tree is a binary tree where each node is either present a version (called version split history or vsh) or present a key (called key split history or ksh). As the name suggested, those nodes in the indexing are the history of the splits. A vsh is caused by a version split, which leads the search according to the whether there is an ancestorship between the version in the vsh and the version queried. A ksh is caused by a key split, which leads the search of according to the key value as the normal B-Tree does. Follow the indexing tree, a search with specified version and key will be fast located into one data page. Figure 2.11 shows a sample of BT-Tree.

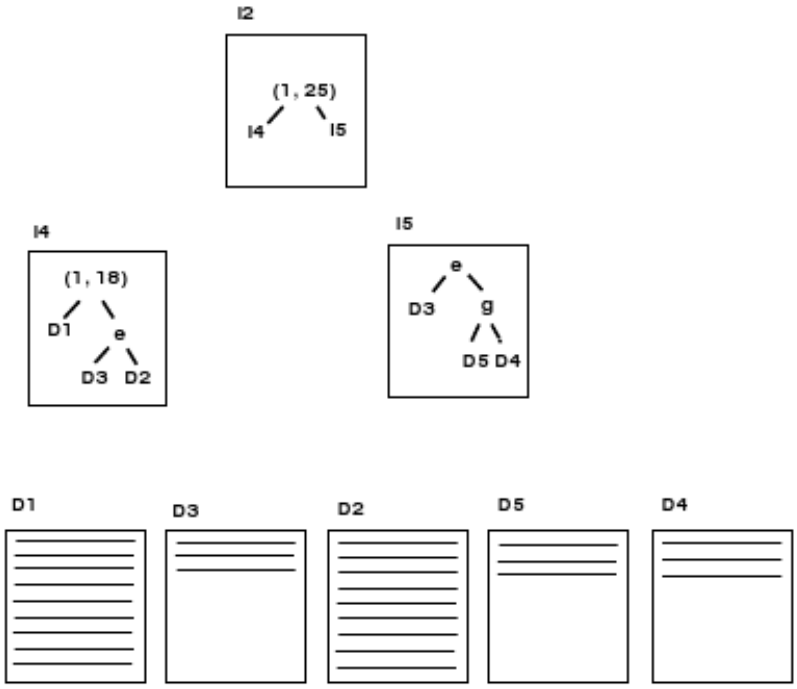


Figure 2.11: A sample BT-Tree

Figure 2.11 shows a BT-Tree with three index pages $I2$, $I4$ and $I5$, and five data pages, from $D1$ to $D5$. $I2$ is the root for the BT-Tree, which leads the search of version that is a descendant of version (1, 25) to the index page $I5$ and those not to $I4$. $I5$ leads a search for key less than e to $D3$, and those greater or equal to e but less than g to $D5$ while those who are greater or equal to g to $D4$.

BTR-Tree [9] used the same main idea, though proposed a new splitting algorithm, namely R-splitting. It decreased the amount of branching in pages while maintaining simple posting and searching algorithms. Results show that the BTR-Tree improves the space performance significantly.

2.4 Querying of XML Data with Multiple Versions

There is little research on the twig pattern matching over XML documents with multiple versions. Zografoula Vagena, et al. proposed an algorithm for path queries over XML documents with branched versioning in [14], which used a variation of BT-Tree [8] to manage the data and use PathStack [6] to do the path query process, though twig query was not supported. Adam Woss combined several existing method to solved the twig queries over linear time evolution XML documents in [16]: MVBT-TwigStack uses MVBT [4] to hold the temporal data and TwigStack [6] to do the twig pattern query process; MVBT-LCS uses MVBT combined with LCS-TRIM [13] to do the sequence-based XML query matching; and TLCS (Temporal LCS) relies on a timestamp XML data model. Those approaches work fine for linear time evolution, though branched versioning XML data are not supported.

Chapter 3

Approach

Our approach focuses on combining modern holistic matching algorithm with the branched versioning data management to operate the twig pattern query process over XML documents. Though separately, there are approaches for holistic twig query matching and branched versioning data management, changes are still necessary to be made for them to work together. We propose a BT-List method to hold the branch versioning data, which is a variation of BT-Tree [8] according to the purpose of XML query process. Then, we combine the BT-List with the XML query process methods and proposed two approaches, BT-TwigStack and BT-LCS.

3.1 BT-List

First consider the requirements for the branched versioning data management in our problem. The query specifies a version time and also the twig pattern. All the data at the query version time should be extracted from the database and put to the XML query

process part to do a static twig matching. Consider the time evolution, some data can be shared across different versions. For example, Version 2 inherits the data entry e from Version 1. To get all the data for Version 2 at some time t, that data entry e, though was created in Version 1, needs also to be extracted. So, a query for all the data at version v, time t requires the accesses to all the data of every ancestor version of (v, t). Considering in the view of a version tree, we need all the data on the path from the root (Version $(1, 1)$) to the version (v, t).

BT-Tree [8] is an effective method to store and query branched versioning data, where usually the query specifies both the version range and the key value, so using the indexing tree, a search can fast locate a data page to get the result. Because in our query, there is no key specified, but the data for all the keys should be extracted, BT-Tree doesn't guarantee a good performance. Because BT-Tree has version split and version-and-key split, data are gathered together if they are close either in version values or in key values, though what we want is the data gathered together only if they are close in version values. Consider the difference, we proposed a variation of the BT-Tree, namely BT-List.

As the name suggested, BT-List focuses on the lists of keys and data of the same version as query results, and it doesn't have a tree to index the data. In BT-List, we have lists of pages for different versions, so when querying a specified version, we can get that list of pages for that specific version and create the results. We add pointers to each data page (now called BT-Page) to indicate the next pages of the lists for different versions. And each BT-List keeps all the pointer to the first BT-Page for each branch. With a queried version, we can follow the pointers to get that list of BT-Pages that contain the data of that

version. Consider the data shared by different versions, the pointers of different versions maybe point to the same page.

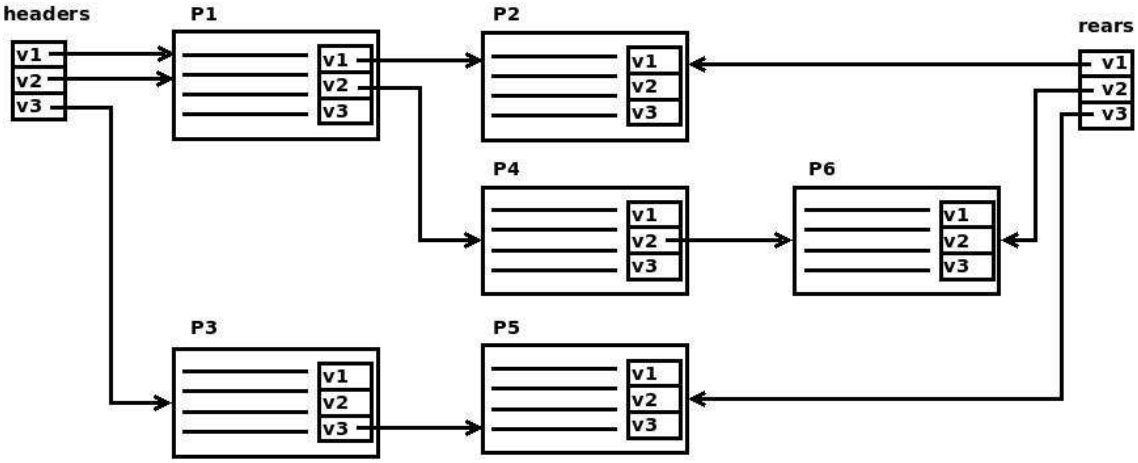


Figure 3.1: A sample BT-List

Figure 3.1 shows an example of BT-List. A BT-List has some headers which are pointers to the first BT-Page of each version. Each BT-Page has pointers to the next page for each version. BT-Page P1 is shared by both Version 1 and Version 2. Querying Version 2 of the time t, it will follow the pointers for Version 2 and access BT-Page $P1$, $P4$ and $P5$. We also keep some rear pointers to the last page of each version. Because the temporal data are inserted always to the leaves of the version tree as the time progresses, i.e. to the last page of the list of a version, a fast locate to the rear of each version is required. Only when new branch in the version tree is created, which would occur in the middle of the version tree, we need to follow the pointers to locate the specific page for the new branch to start.

Data entries in the BT-Page are in order of the time (as they are inserted). One BT-Page may contains data entries of different versions, though these versions must be

connected, called a version range. A version range is a connected part in the version tree with only one start version and a set of end versions, which can be none or multiple versions. The version range contains some branches that are descendants of the start version, and end versions indicate that whether or not one branch is closed to the current time.

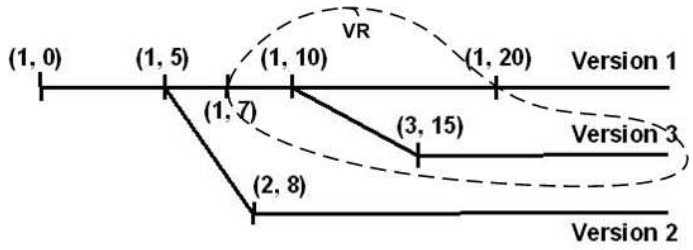
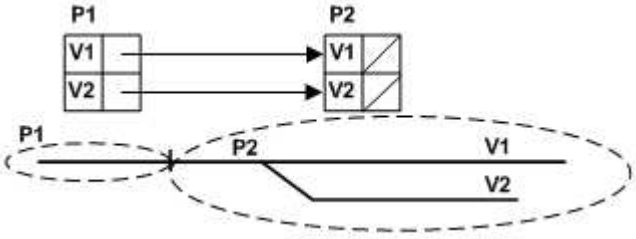


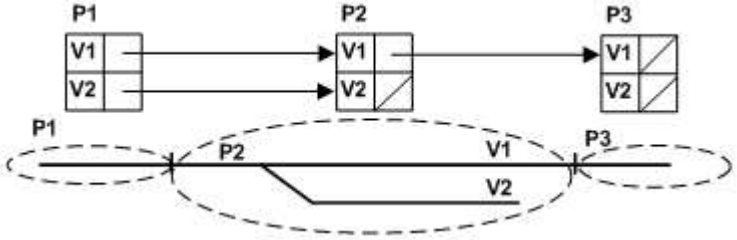
Figure 3.2: A sample of version ranges

Figure 3.2 shows an example of some version ranges. The version range in the middle that is circled by dashed line has the start version $(1, 7)$ and only one end version $(1, 20)$. The version that is a descendant of the start version but not descendant of any end version belongs to the version range. This version range is closed to Version 1 after time 20, but open to Version 2, because there is no end version of Version 2.

The data are inserted into the BT-List in the order of the time. When a BT-Page is overflow, we use some simplified splitting strategy of those in BT-Tree [8] or BTR-Tree [9]. Every BT-Page is owned by one version, though it may contain data for multiple versions. The owning version is not necessary the same version as the start version. We will explain this owning version in detail later. The first case of the split is that a split caused by a new data entry on the owning version of the page will be a current version split. Figure 3.3 shows an example of this situation.



(a) P2 is full before split

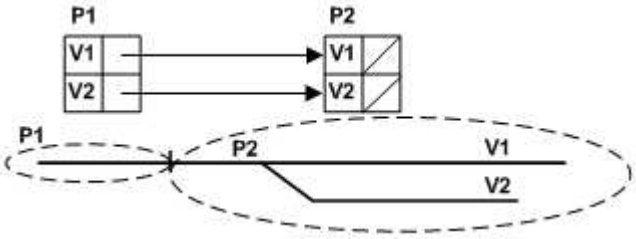


(b) After the split by a new data entry to Version 1

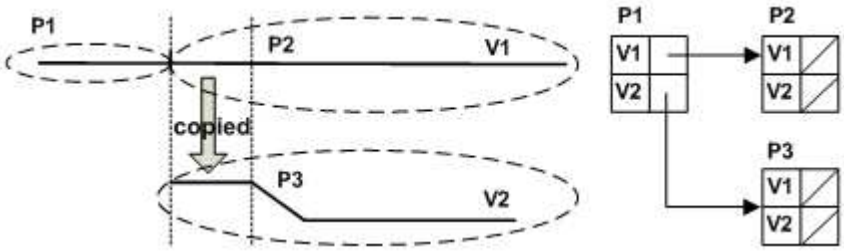
Figure 3.3: Splits caused by the owning version of a page

We use the version tree to indicate the relationship of the versions and the version ranges are circled for each BT-Page, with the pointers in the BT-Pages also illustrated in the figure. Figure 3.3(a) shows the BT-List before the split, where $P2$ is already full. A new data entry of Version 1 is then inserted. It should be put into BT-Page $P2$, which causes an overflow. Because Version 1 is the owning version of $P2$, we use a simple split at the current time, meaning that only the new record is put into a new page, only pointers and the end version set will be change while anything else remains the same. Figure 3.3(b) shows the BT-List after the current time split. A new page $P3$ is created which contains only the new data entry of version 1. The old page $P2$ is closed for Version 1 at the current time, and a pointer to the new page $P3$ is added for $P2$ indicating that the next BT-Page to access for the data for Version 1 is $P3$.

The second case of the split is that a split caused by a new data entry on the non-owning version path of the page will be a split at the born time of that non-owning version. All the data of the descendant version of the born version is *moved* into the new BT-Page. Depends on the number of data entries that is of the ancestor versions of the born version, we will decide whether to copy those data to the new page or not. Figure 3.4 and Figure 3.5 illustrate the two situations in this case.



(a) P2 is full before split

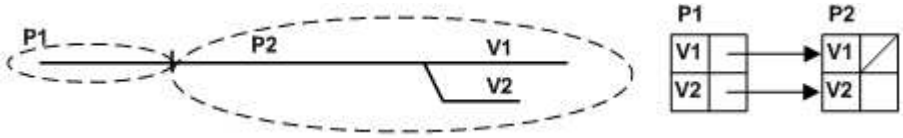


(b) After the non-owning version split with some data copied

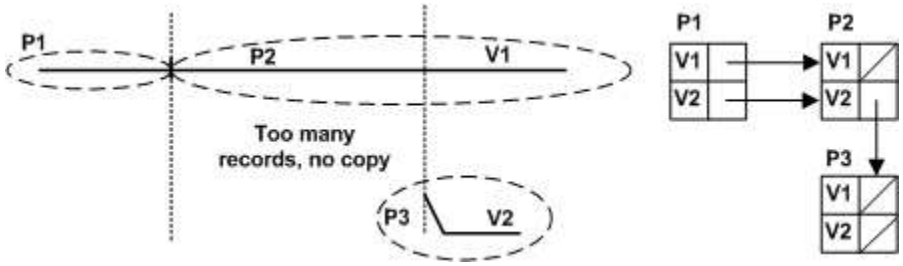
Figure 3.4: Non-owning version splits with data copied

Figure 3.4(a) shows the BT-List before an insertion. $P2$ is already full and the owning version of $P2$ is $V1$. A new data entry of $V2$ is inserted, so a non-owning version split will be occur. All the data of the descendant versions of the born version of $V2$ is moved into the new BT-Page $P3$. Now to access all the data of Version 2, we need to access

$P1$, $P2$ and $P3$. However, the needed data in $P2$ are only from the start version of $P2$ to the born version of $V2$. If there are not so many records of them (less then some threshold), it's better we copy them to the new page, so $P2$ is no longer required to access to get the data for $V2$. Figure 3.4(b) illustrates this situation. The data from the start version of $P2$ to the born version of $V2$ are copied to the new page $P3$, so now to access all the data for $V2$, only $P1$ and $P3$ need to be accessed, but not $P2$. The pointers in the pages are also modified to indicate this. Now, though the start version of $P3$ is the same of that of $P2$, which belongs to $V1$, the owning version of $P2$ is still $V2$.



(a) P2 is full before split



(b) After the non-owning version split without any data copied

Figure 3.5: Non-owning version splits without any data copied

If there are too many records from the start version of $P2$ to the born version of $V2$, they are not copied to the new page. If we do so, its a waist of space having the same copies and time will be spent to copy them; the new data page is more quickly to be full, and more split will be occur; and there is not much time saved in the querying. Figure 3.5

illustrates this situation. In Figure 3.5(b), the new page $P3$ only contains the data since the born of $V2$. To access all the data for $V2$, the pages $P1$, $P2$ and $P3$ will be accessed. $P3$ is clearly owned by $V2$ in this situation.

In all, BT-List is a variation of BT-Tree which focus on the function to get all the data for some specified version. The tree indexing is not necessary here because there won't be much query to the middle of the version tree. Instead, pointers are used to make a sequence scan of all the data for a specified version fast. Different split strategies are used for different situation to make the storage and querying for the branched version data more efficient.

3.2 BT-TwigStack

As the name suggested, BT-TwigStack uses BT-List to hold the data with branched versioning and TwigStack to do the twig pattern query process. TwigStack [6] is a stream based algorithm, to process a twig pattern query, the list of all the occurrences of each element in the query is needed. So in the BT-TwigStack, the occurrences of each element are stored in separate BT-List, i.e. one BT-List stores the data for one element. As mentioned in Section 2.1, the labeling scheme ORDPATH supports the dynamic updates of XML documents. In BT-TwigStack, ORDPATH is used to represent the position of each occurrence of each element. So the data stored inside BT-List are the ORDPATH labels.

To process a query, BT-Lists of the elements in the twig will extract all the ORDPATH labels of all occurrences of those elements at the querying version. For each element, those ORDPATH labels are sorted according to their position in the XML document. Then,

those sorted ORDPATH lists of elements are pushed to TwigStack, where a twig pattern matching can be done as if it is a static query process. It's easy to check the position relationship between two XML nodes by their ORDPATH labels, which makes it easy to sort the list of data before pushing to the TwigStack.

3.3 BT-LCS

Also as the name suggested, BT-LCS uses BT-List to hold branched versioning data and then LCS-TRIM approach is used to do the twig pattern query process. The main idea of BT-LCS is similar to our BT-TwigStack: we stored and indexing the branched versioning XML data using BT-List; when a query came, we get the XML document of the queried version and do the twig pattern matching.

As one of the sequence based techniques, LCS-TRIM [13] requires both the XML document and the twig query be first converted into Consolidated *Prüfer* Sequence (CPS), so that subsequence matching (using the dynamic programming technique for the Longest Common Subsequence problem) and structural refinement can later be carried on. A CPS consists of two sequences, Numbered *Prüfer* Sequence (NPS) and Label Sequence (LS). To construct the CPS for a XML document, we need to first number each element as in the post-order, then still in the post-order, we record the numbers of their parents (to construct NPS) and the label of itself (to construct the label sequence LS). Recall that the data we stored in the BT-List are the ORDPATH labels of all the XML nodes, we need to convert this list of ORDPATH labels of the nodes into CPS before we send the data to LCS-TRIM. Because the structural relationship of two nodes can be easily told from their

ORDPATH labels, we reconstruct the XML document as in the tree structure and create the corresponding CPS.

In LCS-TRIM, a good optimization is called Label Filtering, which will first prune from the elements (or called labels there) not appeared in the twig query. As the number of distinct elements in the data is usually a lot higher than that in the query, this optimization improved the algorithm a lot in both time and space complexity.

In our BT-LCS approach, we store the temporal XML data in BT-Lists, one for each element. This makes the Label Filtering optimization also available in the BT-LCS: the BT-Lists of only the elements in the twig query are accessed, and only those XML nodes are retrieved instead of the entire XML document at the queried version. This will decrease a lot the size of the sequences we push to LCS-TRIM algorithm, though it also brings a problem: we get separate XML nodes which are not necessarily connected (we still need those unconnected nodes to represent the structural relationship between nodes), and this also brings problem to create the *Prüfer* sequences. It doesn't matter what real nodes are, because they are not elements queried. We need only to preserve the structural relationship of the nodes we need. To solve this problem, we insert pseudo nodes just to connect the nodes, with some faked element label. Now, with this legal XML document, we can create the *Prüfer* sequences and push to LCS-TRIM algorithm to do the twig pattern matching.

Chapter 4

Results

In this chapter, we experimentally evaluate our algorithms: BT-TwigStack and BT-LCS, comparing with the traditional log-based approach as a baselines. In Section 4.1, we discuss the settings of the experiments, where in Section 4.1.1 and Section 4.1.2, we explain the data set and query set we used in the experiments respectively. In Section 4.2, we compare the experimental result using different approaches for the twig queries over XML Document. We consider the two parts of the algorithm: indexing process (in Section 4.2.1) and query process (in Section 4.2.2) separately.

4.1 Experimental Settings

All the experiments were performed on a system with an Intel Pentium 4 1.60GHz CPU and 1 GB of main memory.

4.1.1 Data Set

We used the XMark [1] benchmark to generate the synthetic data sets for our experiments. xmlgen [2] produces XML documents modeling an auction website, a typical e-commerce application. We used a 600MB XML documents with more than 7 million nodes.

We have created a simple program to simulate the time evolution of the XML documents by applying a batch of inserts, removes and having more branches of version evolutions. A new version branch is created occasionally inherit from a random historical version. New XML nodes are inserted into random version branch that already exists. And randomly we will have some existing node to be deleted. We have a 10% ratio for the removal of the existing nodes. Each of these modification creates a new version in the database, though in the reality, we can also have several modification to create a new version. Though the simulation strategy makes the data randomly, we carefully guaranteed that the data are fully legal and reasonable. Each version tend to have as many nodes as each other. We use two data sets with branched versioning. $D1$ has 20 branches with 5 million time points and $D2$ has 20 branches with 7 million branches.

4.1.2 Query Set

For the structural queries over the XML documents with branched versioning, a query is composed with a twig pattern and a version. The twig patterns are syntactically similar to XQuery [5] notation, while the version needs to give both the version id and the time. Table 4.1 lists the twig pattern part of the queries we used in our experiments. For

each twig pattern, we queried the latest time of the versions spanning different number of versions.

Query id	Query Expression
Q1	`//item[//quantity="10"][//location="United States"]`
Q2	`//mailbox/mail[from="customer" AND to="service"]`
Q3	`//bidder[//date="6/12/2007"][//time="12:00"][//increase="7.5"]`
Q4	`//item[//mail/date="12/25/2007"][//payment="Credit Card"]`

Table 4.1: The twig patten queries used in the experiments

4.2 Experimental Results

In this section, we conduct several experiments and compare the results. We divide the entire temporal twig query process into two parts: indexing process and query process. In the indexing process, no matter what kind of indexing method is used, the temporal XML data are read and parsed and stored in some particular format for later temporal twig query process. This process is only related to the data, but not queries. The second part is the query process, where those stored temporal data are access to do the twig pattern matching for particular versions. We analysis the experimental results for each parts separately.

4.2.1 Indexing Process

In this section, we consider the indexing part of the algorithms. This part is separated from the parts of the query process. In this indexing process, BT-TwigStack and BT-LCS are reading the temporal XML documents and creating the BT-Lists. BT-TwigStack and BT-LCS use the same BT-List technique as the indexing process, both

store different elements of the XML document into separate BT-Lists. Though log-based approach doesn't have any indexing for the temporal data, it still needs time to read and parse the temporal data and store in its own format for later use. We also use this time as the baseline of the I/O of reading and parse the temporal data, to compare with the time cost of BT-TwigStack and BT-LCS, which have both the I/O part and the construction of the BT-List.

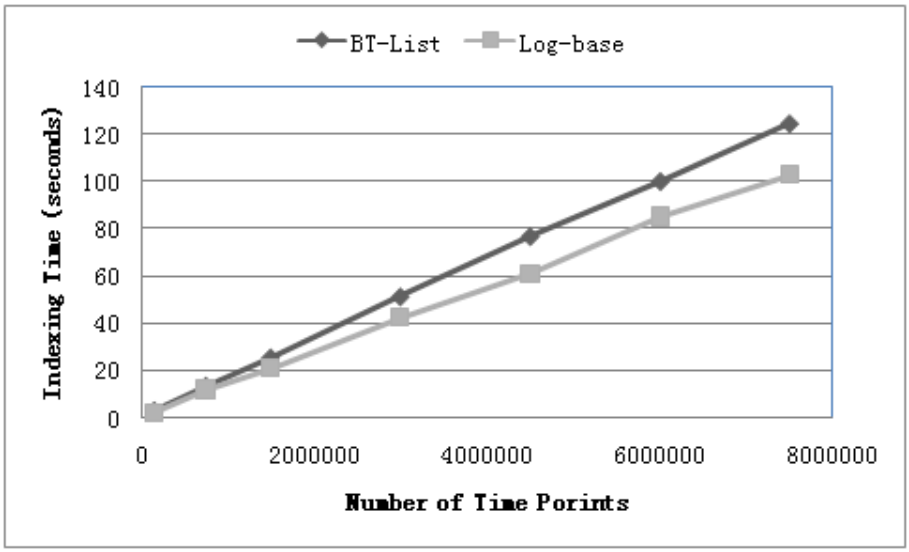


Figure 4.1: Indexing time used for different algorithms

Figure 4.1 demonstrates the time cost to build the BT-Lists for BT-TwigStack and BT-LCS, compare with the I/O time for log-based, as the XML document grows in size. We can see that it is very fast to construct BT-List, which is only about 20% of the time of reading and parsing the same amount of temporal data. For a 500 MB XML documents with 7.5 million time points across 20 versions, the indexing process can be done in about 2 minutes.

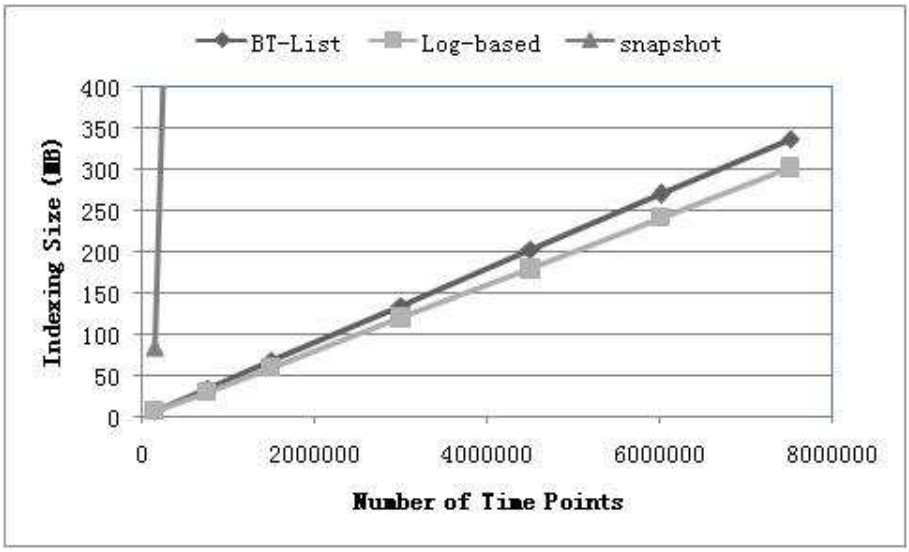


Figure 4.2: Storage usage of different algorithms

Figure 4.2 demonstrates the space used to hold the temporal XML data: BT-Lists for BT-TwigStack and BT-LCS, the plain log for the log-based approach and we even compare them with the snapshot indexing. From the figure, we can clearly see that the log-based approach requires the minimal space usage, which because of no extra overhead for any structural managements or indexing. The overhead for BT-Lists is mainly caused by the extra copies of some data which makes the access of data more efficient. We can also see that the overhead is within a very reasonable amount. On the other hand, the snapshot method keeps each version separately, which waste a lot of space storing many copies of the same data that has not been modified. We can see that snapshot exhausts the memory very quickly.

4.2.2 Query Process

In this section, we consider the time cost for the temporal twig query of the three algorithms. To do the twig query on a specified version of the data, the related XML data need to be first retrieved (preprocess) and then the data are used in the XML twig matching techniques.

In the preprocess, the log-based approach needs to reconstruct the XML document at the queried time by redoing every modification of the time evolution. Though the reconstruction is only about the elements in the twig pattern, not necessarily the entire XML document, all the log entries up to the time of the version queried need to be accessed. If there are many versions in the database, especially when many of them doesn't affect the queried version (is not an ancestor version of the queried version), this process requires more time, because a lot of unrelated data has to be accessed. The log-based approach here is also using TwigStack technique for the twig query matching part, so before the list of ORDPATH labels is pushed to TwigStack, it has to be sorted in order as required by the TwigStack algorithm. Instead of first getting the list then sort it, we keep the list in order as getting each occurrence of the elements.

The BT-TwigStack and BT-LCS ask the BT-List about the data at the version queried. In both the two algorithms, different elements are stored into different BT-Lists. So only those BT-Lists of the elements required in the query needs to be accessed. As mentioned above, TwigStack requires the occurrences of the elements to be sorted, so we need to sort the ORDPATH labels in BT-TwigStack for each elements. On the other hand, LCS-TRIM needs the XML documents to be given in the form of Consolidated *Prüfer*

Sequence (CPS), so one more step is needed for BT-LCS to reconstruct a part of the XML document and create the corresponding CPS.

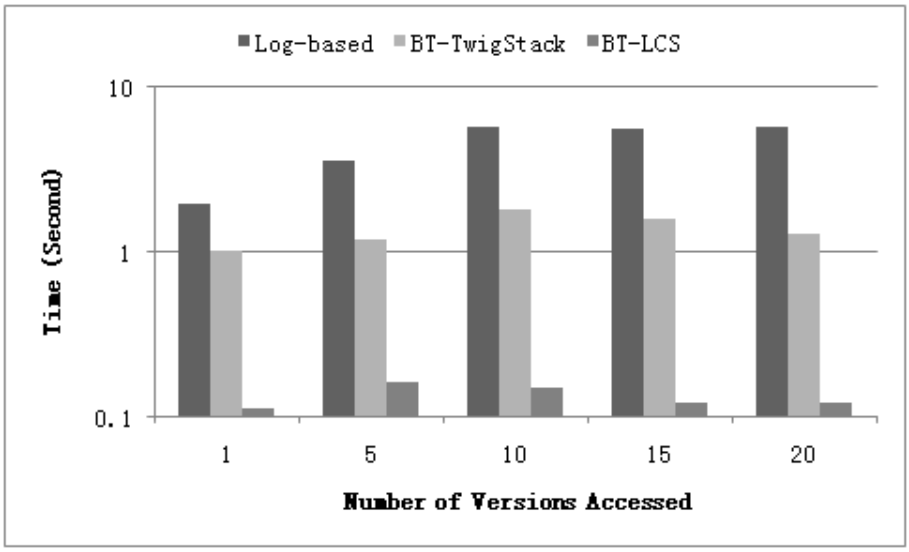


Figure 4.3: Execution time of Q1 across different number of version branches for D1

Figure 4.3 - 4.6 demonstrates the time cost for the twig query process for $Q1$ - $Q4$ across different number of versions at the time point of 5 million for $D1$.

First we consider the data accessed by the query. The log-based approach spends a huge amount of time accessing all the temporal data up to the queried time point. BT-TwigStack and BT-LCS which are using BT-List access only the data of the elements in the twig query and affects to the queried version (belongs to an ancestor version) Compared with the log-based approach, BT-List eliminates the access to the data of two kinds: first kind is the data which do not belong to an ancestor version of the queried version; and second is the data of the elements that do not appear in the twig query. Both eliminations guarantee that no solution is false deleted from the result. The less data to be accessed,

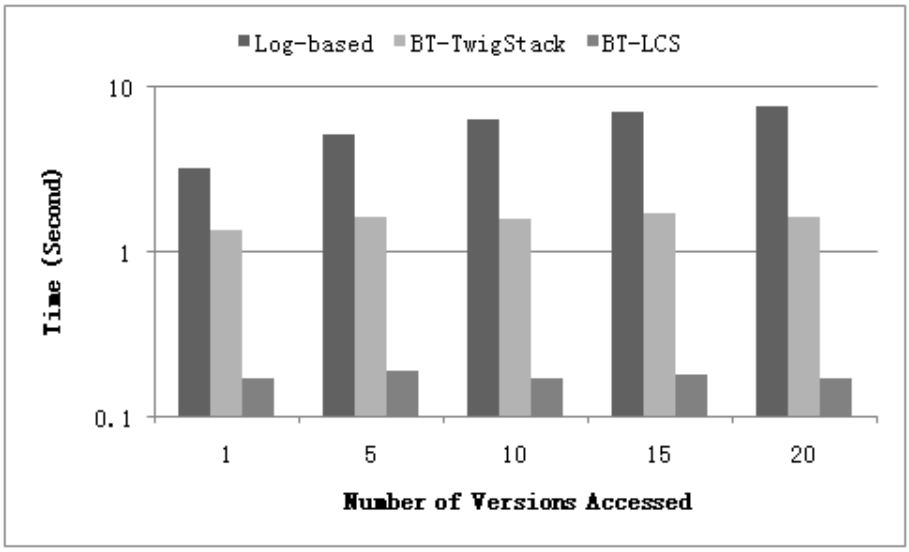


Figure 4.4: Execution time of Q2 across different number of version branches for D1

the less time it costs to do the query process. These eliminations make the algorithm much more efficient.

Compared the BT-TwigStack with BT-LCS, they both use the same BT-List to hold the temporal data, so they needs the same amount of time to access the BT-Pages to get the XML nodes. However, the difference is, in BT-TwigStack, the XML nodes need to be sorted as they appear in the pre-order of the XML tree by only comparing the ORDPATH labels; while in BT-LCS, the XML tree need to be constructed, so that CPS can be computed from the tree structure. In BT-TwigStack, we use a sorted list for the ORDPATH labels for each element, at the same time we find a temporal data entry, we insert them at the correct position so that the list is kept sorted, so it cost some time to find the correct position to insert the data and possible moves of the data would also be carried out. In the BT-LCS, as soon as we get a temporal data entry, we put it into the

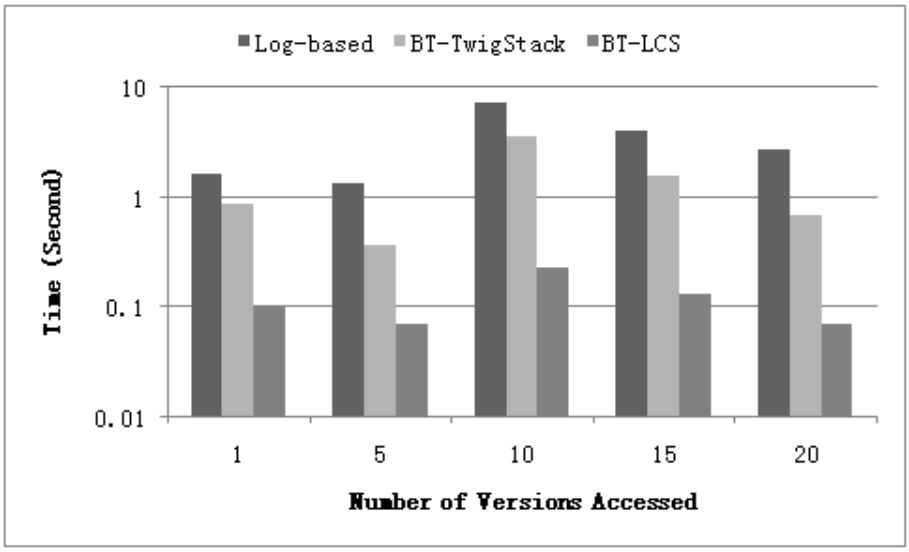


Figure 4.5: Execution time of Q3 across different number of version branches for D1

tree according to the ORDPATH label. After the tree is constructed, we need one more step to number the tree nodes according to the post-order and construct the CPS.

Figures 4.3 - 4.6 shows that BT-TwigStack and BT-LCS finishes the query much faster than the log-based approach and BT-LCS almost always beat BT-TwigStack.

Figure 4.7 - 4.10 shows the execution time of several queries over a much larger data set which contains about 7 million time points and 20 versions for $D2$. Still from those figures, we can see that BT-TwigStack and BT-LCS beat the log-based approach dramatically and BT-LCS is almost always better than BT-TwigStack.

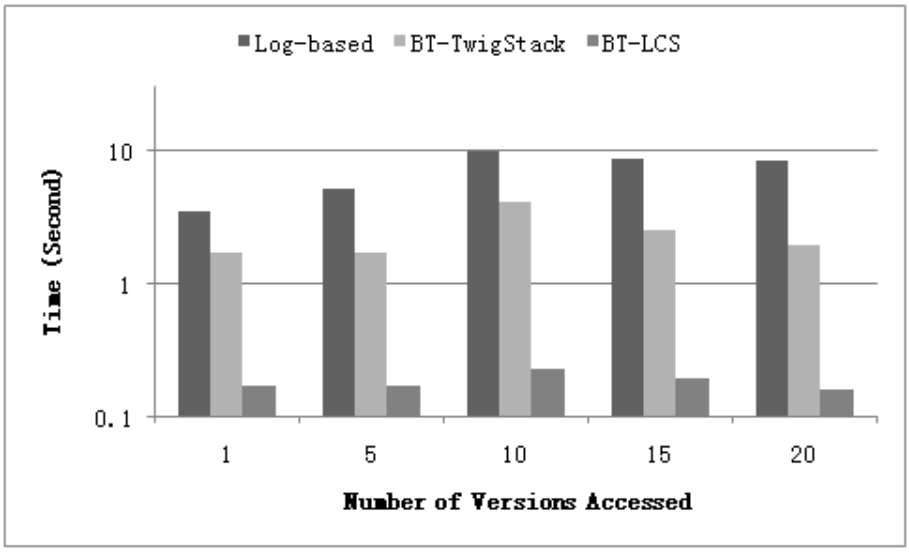


Figure 4.6: Execution time of Q4 across different number of version branches for D1

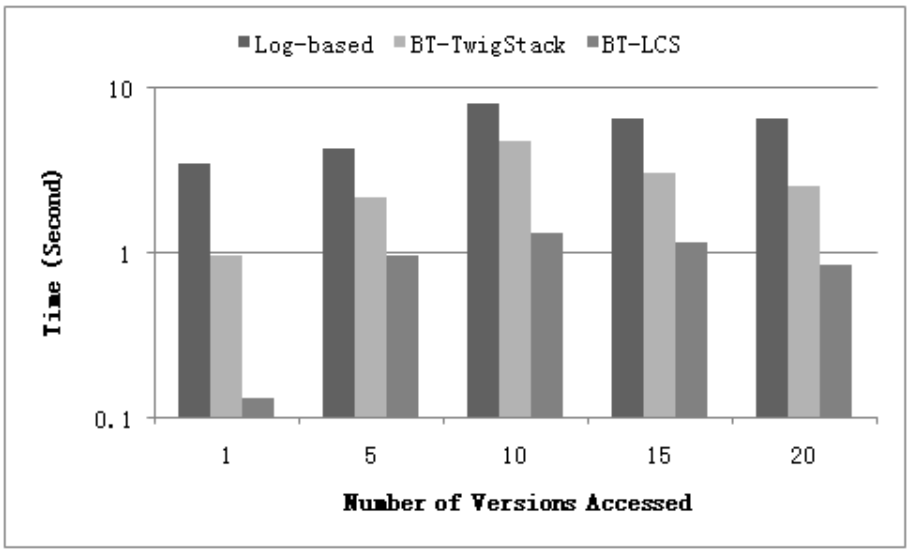


Figure 4.7: Execution time of Q1 across different number of version branches for D2

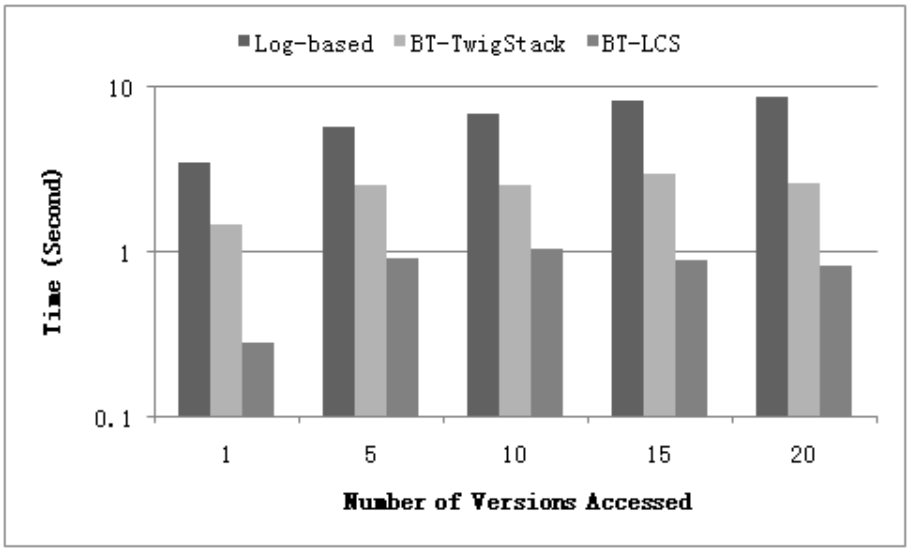


Figure 4.8: Execution time of Q2 across different number of version branches for D2

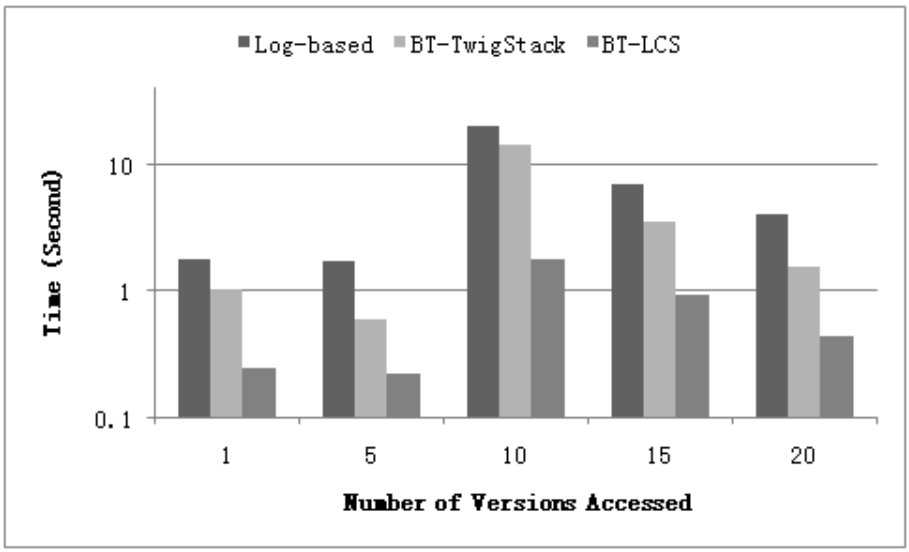


Figure 4.9: Execution time of Q3 across different number of version branches for D2

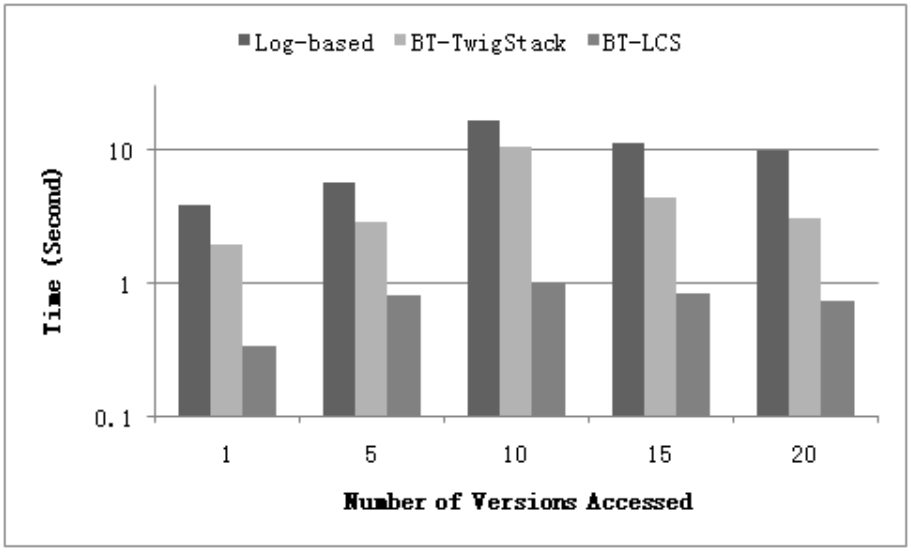


Figure 4.10: Execution time of Q4 across different number of version branches for D2

Chapter 5

Conclusion

As XML usage continues to increase over the last decades, so does the importance to preserve the historical information of the documents. Some great amount research interests have been paid on the problem of twig pattern query matching on the XML documents, though most of them are done for static XML documents. On the other hand, a significant amount of work has focused on the managing and indexing of the temporal XML documents, not only with the linear versioning data (where only the latest version can be updated), but also branched versioning data (where both a historical version and the latest version cane be modified, i.e. the version evolution is a tree with branches). Nevertheless, the problem of querying the temporal XML document, especially the XML documents with branched versioning has been relatively untouched.

In this thesis, we have proposed several algorithms for the structural queries over temporal XML document with branched versioning. More specifically, we expended, modified and combined the current state-of-the-art techniques, for both static XML document

query process and temporal XML document indexing and management, to work together. We first choose the ORDPATH as the labeling scheme for the XML documents to enable efficient dynamic modifications in the sense that no tree-relabeling is required. We proposed a BT-List approach to hold and index the data with branched versioning, which is later integrated with two kinds of XML query matching algorithms, TwigStack and LCS-TRIM as our overall algorithms for the problem: BT-TwigStack and BT-LCS. The experimental results show that BT-List can efficiently index the data which required reasonable space and time. The temporal twig query can be fast matches using our BT-TwigStack and BT-LCS algorithms compared with log-based approaches.

Bibliography

[1] XMark: An XML benchmark project, 2003. `http://www.xml-benchmark.org/`.

[2] xmlgen: The benchmark data generator, 2003. `http://monetdb.cwi.nl/xml/generator.html`.

[3] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–, 2002.

[4] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[5] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. Technical report, W3C, January 2007.

[6] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[7] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[8] Linan Jiang, Betty Salzberg, David B. Lomet, and Manuel Barrena García. The bt-tree: A branched and temporal access method. In *VLDB*, pages 451–460, 2000.

[9] Linan Jiang, Betty Salzberg, David B. Lomet, and Manuel Barrena García. The btr-tree: Path-defined version-range splitting in a branched and temporal structure. In *SSTD*, pages 28–45, 2003.

[10] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD Conference*, pages 903–908, 2004.

[11] Praveen Rao and Bongki Moon. Prix: Indexing and querying xml using prüfer sequences. In *ICDE*, pages 288–300, 2004.

[12] Betty Salzberg, Linan Jiang, David B. Lomet, Manuel Barrena García, Jing Shan, and Evangelos Kanoulas. A framework for access methods for versioned data. In *EDBT*, pages 730–747, 2004.

[13] Shirish Tatikonda, Srinivasan Parthasarathy, and Matthew Goyder. Lcs-trim: Dynamic programming meets xml indexing and querying. In *VLDB*, pages 63–74, 2007.

[14] Zografoula Vagena, Mirella Moura Moro, and Vassilis J. Tsotras. Supporting branched versions on xml documents. In *RIDE*, pages 137–144, 2004.

[15] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. Vist: A dynamic index method for querying xml data by tree structures. In *SIGMOD Conference*, pages 110–121, 2003.

[16] Adam Woss. Twig Queries Over Multiversion XML Documents. Master's thesis, University of California, Riverside.