Twig Pattern Search in XML Database

By

LEPING ZOU

A thesis submitted to the

Department of Applied Computer Science

in conformity with the requirements for

the degree of Master of Science

University of Winnipeg

Winnipeg, Manitoba, Canada

December 2010

Abstract

Today, Extensible Markup Language (XML) is becoming more and more popular for data representation and data exchange over the World Wide Web. So, more data files over the WWW will be represented in the XML format; and handling a large amount of XML documents becomes compelling. For the current search technology, we often have the experience that we can find many results when we search the Internet by issuing some key words, but most of them are useless or just not the one we want. So, for the next generation of the search engine, the main challenge is how to find what we exactly want. The main purpose of this thesis is to develop an algorithm for efficiently searching a pattern, called a *twig pattern* or *tree pattern*, to find all the matching documents. Unlike the traditional index methods that split a tree pattern query into several paths, and then stick the results together to provide the final answers, the twig pattern search uses tree structures as the master unit of queries to avoid expensive join operations. In our research, an efficient algorithm for the tree mapping problem in XML databases is proposed. Given a target tree T and a pattern tree Q, the algorithm can find all the embeddings of Q in T in O ($|D||Q|$) time, where D is the largest data stream associated with a node of Q.

Acknowledgments

I am heartily thankful to my supervisor, Dr. Yangjun Chen, for his encouragement, guidance and support from the initial to the final level enabling me to develop an understanding of this thesis work.

I would also like to thank to my friend Sandra Leone, Jian Ren and Buddhika Madduma, who spent their time on the implementation of the algorithm and the performance experiment. I am grateful to their patience and valuable helps.

My deepest gratitude goes to my parents, Yuanming Zou and Zhengwu Shen for their emotional support and encouragement. I dedicate this thesis to them.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of this thesis work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

XML stands for Extensible Markup Language. It is a series of rules for marking up documents in a form which can be understood by computer. The Specification is produced by W3C (World Wide Web Consortium). XML not only describes the data itself, but also the semantics of the document. This enables users to organize information flexibly. That is the reason why it is used so widely in today's Internet. In order to give an intuitively impression, a sample of XML document is given below.

```
< menu>
        <food>
                <name>Belgian Waffles</name>
                <img src=" Belgian Waffles.jpg" alt='by Raphael'> </img>
                <price>$5.95</price>
                <description>two of our famous Belgian Waffles </description>
                <calories>650</calories>
        </food>
</ menu>
```

Figure 1.1 A sample file in XML format

In this sample, a menu record is represented in XML format. It basically contains three components: elements, contents, and attributes. An element is a component begins with a start-tag and ends with a matching end-tag, such as <menu> and </menu> in the above example. Content is a "raw" data that

represents the content of a document such as "Belgian Waffles". An attribute is a name/value pair that represents the additional properties of an element. For example, the element *img* has two attributes: *src* and *alt*, specified as follows

<img src=" Belgian Waffles.jpg" alt='by Raphael'> </img>.

Usually, an element can contain content and sub-elements, i.e. multiple elements which can be nested in some way. Therefore, any XML document can be represented as a tree-like structure, referred to as a document tree or an XML tree, in which all contents are mapped to the leaf nodes and all element tags are mapped to the internal nodes. For example, Figure 1.2 shows the tree structure associated with the sample document shown in Figure 1.1.

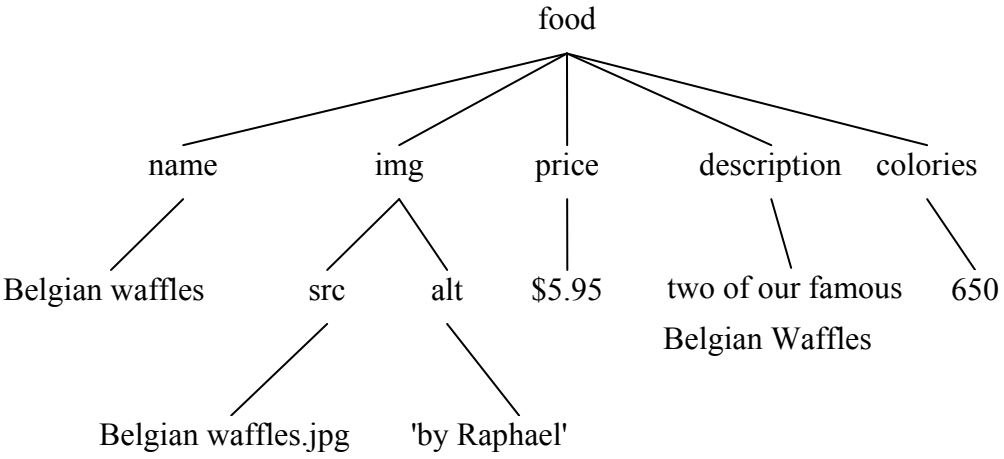


Figure 1.2 XML data structure shown as a tree

1.1 Document Type Definition (DTD)

A DTD describes the structure of a class of XML documents by the element and attribute-list declarations. In an element declaration, the names of all its sub elements are given, such as *menu* containing *food*, and *food* containing *name*, *img*, *price*, *description*, and *calories*, which will be put in a pair of parentheses as shown in the above example. Attribute-list declarations name the possible set of attributes for each element, such as the #PCDATA followed by the name of the element.

```
<?xml version="1.0"?>
<!DOCTYPE menu[
    <!DOCTYPE food [
        <!ELEMENT menu (food)>
        <!ELEMENT food (name, img, price, description, calories)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT img (src, alt)>
        <!ELEMENT price (#PCDATA)>
        <!ELEMENT description (#PCDATA)>
        <!ELEMENT calories (#PCDATA)>
        <!ELEMENT src (#PCDATA)>
        <!ELEMENT alt (#PCDATA)>
    ]>
]>
```

Figure 1.3. A sample of DTD

1.2 Problem Statement

The XML is a tree-structured model for representing data. As more and more XML files are widely used in the Internet for data exchange and storage, searching in XML becomes important. One of the methods is the tree pattern matching, heavily used in the systems offering search in XML Languages such as XPath [15] and XQuery [14]. XPath is a declarative language, and XQuery is an iterative language which uses XPath as a building block, providing path expressions as a searching condition. For example, /food/img/[alt = 'by Raphael'] is a path expression that inquiries one of the paths in the tree shown in Figure 1.1(b) to find any picture painted by Raphael. Multiple path expressions can form a complex query that contains multiple paths, which are in fact a tree structure. Ordinarily, the query tree is small. So, the corresponding tree matching problem is called a "twig" pattern matching. In Figure 1.4, we show a simple twig.

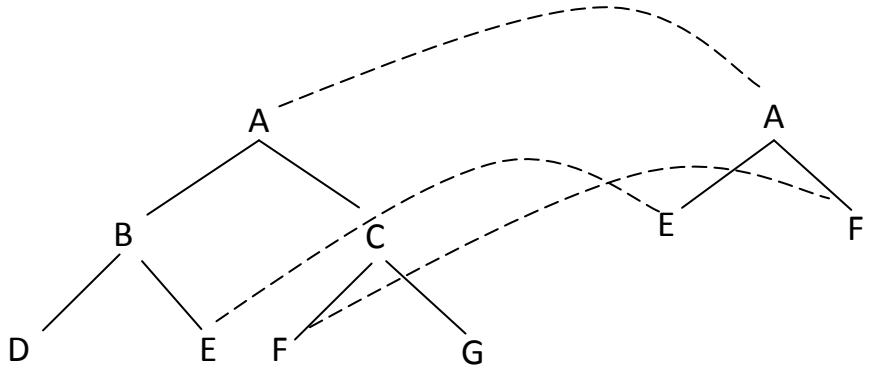


Figure 1.4 A sample of tree embedding

Definition [5] An embedding of a tree pattern Q into an XML document T is a mapping f: $Q \rightarrow T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) Preserve node label: For each $u \in Q$, $label(u) = label(f(u))$ (or say, u matches $f(u)$).

(ii) Preserve *parent-child*/*ancestor-descendant* relationship: If $u \rightarrow v$ in Q, then $f(v)$ is a child of $f(u)$ in T; if $u \Rightarrow v$ in Q, then $f(v)$ is a descendant of $f(u)$ in T.

If there exists a mapping from Q into T, we say, Q can be imbedded into T, or say, T contains Q.

Up to now, a lot of methods have been proposed to solve this problem. Early methods, such as those discussed in [2, 6, 9], works as follows. First, a twig pattern is decomposed into multiple paths to find the match. Then, all the paths are joined together. This definitely involves the time consuming join operations. Recently, several holistic twig join algorithms are proposed to solve the problem. The first one is *TwigStack* [4]. It uses a stack to handle the intermediate results. Because path matches don't need to be part of complete matches, a lot of redundancy is conducted (we will discuss this in great detail in a later section. The other holistic twig join methods can be found in [7, 10, 16, 17, 18, 19]. Generally speaking, they all try to improve in two directions: improving the join algorithms, such as [7, 10, 16]; and using indexes to speed up accessing disks, such as [17, 18, 19].

In this thesis, we proposed a new algorithm with no join operations involved. The algorithm takes a set of data streams as inputs, and establishes ***XB-tree***

as indexes. By combining these two strategies, we achieve an efficient method for evaluating twig pattern queries.

1.3 Preliminaries

In this part, we will present some concepts related to this thesis, including the definition of trees, tree encoding, and the index structure of ***XB-Tree***. These conceptions are quite necessary for a further discussion.

1.3.1 Tree

A tree structure is a way to represent the hierarchical nature of data, as illustrated in Figure 1.5. The elements are referred to as "nodes", and the lines connecting elements are as "branches. We use *T* to represent a tree and the root of the tree is denoted by $Root_T$. Nodes without children are called leaf nodes. The names of the relationships between nodes come from family relationships. A node *v*, which is one level higher than another node *u*, is called the parent of *u* if they are on the same path. For example, the node c is the parent of e and f. The nodes having the same parent are called siblings. For instance, b and c are siblings. They have the same parent a. The number of a node's children is called the degree of the node. The degrees of node b and c are 1 and 2, respectively. A subtree is a tree whose root is the child of some non-root node. For example, in Figure 1.5 the trees rooted at b and c are two subtrees of node a, which is the root of *T*. There are three important properties for any tree we discussed: *size*, *height*, and *width*. The total number of the nodes is called the tree's *size*. The length of the longest path in a tree is called the tree's *height*. Finally, the number of the leaf nodes is

the tree's width. In Figure 1.5, the size of *T* is 7, the height is 4, and the width is 3.

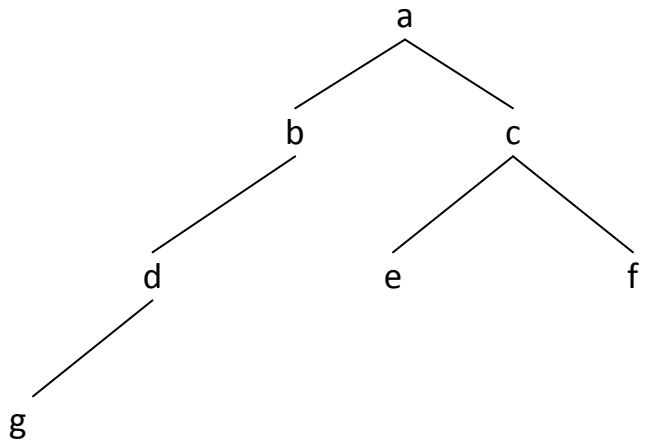


Figure 1.5 A sample of tree structure

1.3.2 Tree encoding

For the efficiency of twig searching, tree encoding schema is very important. It includes two aspects: how the nodes in a partition are ordered, and how the position of a node is encoded. For the first question, most algorithms store nodes pre-orderly by using the depth first traversal. It means that a ancestor can be seen before its descendants. For the second question, a tree encoding is used, which assigns *leftPos*, *RightPos*, *level* values to nodes to recognize their different relationships, as shown in Figure 1.6 (b). The *LeftPos* and *RightPos* numbers reflect the positions of opening and closing tags in XML.

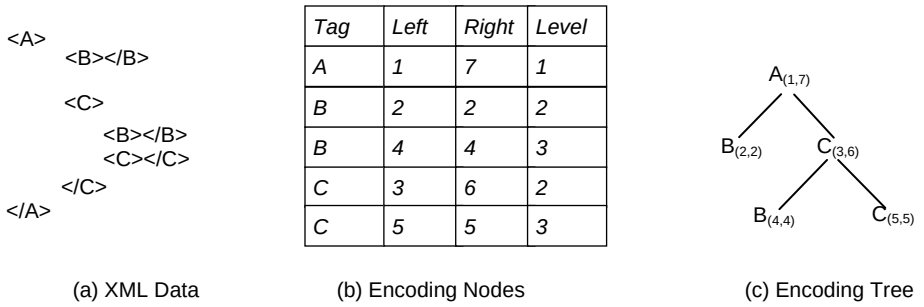


Figure 1.6 A sample of tree index and query

1.3.3 *XB-tree* Index

In our algorithm, we will use an index structure, the so-called ***XB-tree*** [4], to improve the search efficiency. As the name suggests, an ***XB-tree*** is just a variant of B^+-trees. However, an ***XB-tree*** is constructed based on the encoding scheme discussed above.

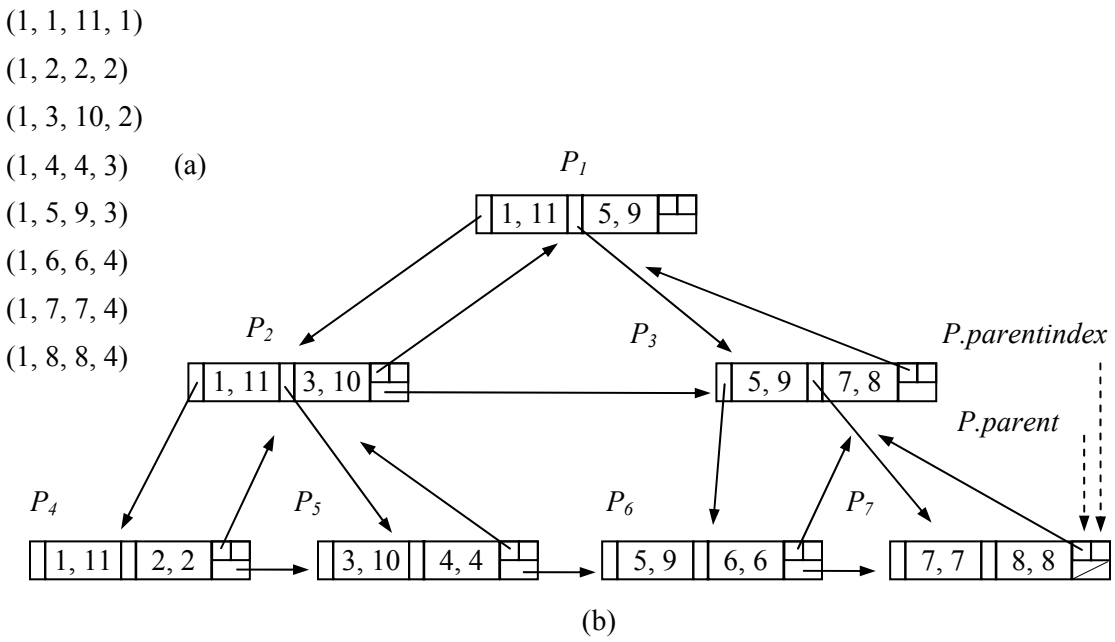


Figure 1.7 A sample of ***XB-tree***

Assume that we have an XML document file stored as a data stream as described in Figure 1.7 (a). We can establish an ***XB-tree*** over it, as shown in Figure 1.7 (b). The nodes in the leaf pages of the ***XB-tree*** are sorted by their *LeftPos* values, and each node is connected by a link from left to right. The main difference between a B^+-tree and an ***XB-tree*** is in the data contained in the internal pages. Each entry v in an internal page of the XB-tree contains a pair $[v_{.L}, v_{.R}]$ (where L and R represent *LeftPos* and *RightPos*, respectively; and the whole pair represents a bounding segment) and a pointer to its child page $v.page$ (which contains all those nodes with pairs completely included in $[v_{.L}, v_{.R}]$ as shown in Figure 1.7 (b). For example, P_2 contains P_4 and P_5. We can also find that all L values in a page are in increasing order although the bounding segments in a page may partially overlap. For instance, in P_2, (1, 11) contains (3, 9). Each page P has a pointer to the parent page, denoted as $P.parent$. In addition, $P.parentIndex$ is an index of the node in $P.parent$, which points back to P as shown in the Figure 1.7(b). We will discuss how to use ***XB-trees*** in the next section.

1.4 Object

The main goal of this thesis is to create a new algorithm for evaluating twig pattern queries, including:

- Implementing a new bottom-up twig pattern search algorithm which can be applied to efficiently determine whether one tree can be embedded in another.

- Investigating the effectiveness of this algorithm and comparing it with other 3 different algorithms which are also used for twig patter search problem.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. In Section 2, we review and discuss the related works. In section 3, we discuss our algorithm in great detail. Section 4 is devoted to the implementation and experiments. Finally, the conclusion and future work are set forth in Section 5.

Chapter 2

Related Work

Generally speaking, the problem of twig pattern matching, is to query all existing embedding patterns in the data. This problem can be classified into two different categories. The first one is the unordered tree pattern, in which only the ancestor-descendant (A-D) and parent-child (P-C) relationships in a twig are considered. The second one is the ordered tree pattern, in which all structural information in the query has to be checked in the data. But the majority of twig queries in practice only concerns A-D and P-C axes. In this chapter, we will review the previous work on this topic.

2.1 The early research for twig pattern matching

The early solutions [2, 6, 9] on the twig pattern matching generally consisted in first decomposing twig queries into binary structural relationships between pairs of nodes, and then matching each of the binary relationships against the XML database. The final results are created by joining together all the path matches.

The main disadvantages of these decomposition-based approaches are that the size of the intermediate results can be very large, even for quite small search results. Another disadvantage is for the P-C relationships. The algorithms work well for treating the A-D relationships. But in the presence of the P-C relationships, a lot of useless matches will be conducted. So, the

users may wait long to get (partial) results. In order to overcome this problem, many interesting twig join algorithms have been proposed.

2.2 Holistic twig join

The first holistic twig join algorithm was *TwigStack*, proposed by Bruno et al [4]. It can be divided into two-phase. In the first phase, all those paths in an XML document will be found, each of which matches a *root-to-leaf* path in the query. In the second phase, they are joined together to form the final result. The core idea of this method is to maintain a stack for each query node.

In general, each query node q in a query Q is associated with matching stream $\boldsymbol{T}_q$; and a stack for a query node q, denoted as $\boldsymbol{S}_{\boldsymbol{q}}$, is used to keep the current ancestor nodes of q. For simplicity, we use a path set shown in Figure 2.1(a) for illustration. Figure 2.1(b) is the query, in which $label(q_1)$ = A, $label(q_2)$ = B, and $label(q_3)$ = C. So, we have T_{q_1} = {A_1, A_2}, T_{q_2} = {B_1, B_2, B_3}, and T_{q_3} = {C_1}. Each data entry in a stack consists of a pair: (positional representation of a node from $\boldsymbol{T}_q$, pointer to an entry in $\boldsymbol{S}_{parent\ (q)}$) as show in Figure 1(c) :

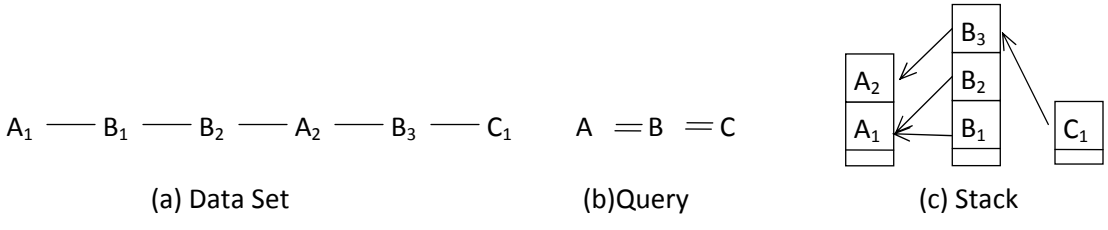


Figure 2.1 Illustration of *TwigStack* process

Figure 2.1(c) shows the stacks for all the query nodes in Figure 1(b).

According to the algorithms discussed in [4], when the current query node is a leaf, all related matching nodes are output. So, when the node C_1 is pushed into the stack, all matching nodes will be popped out. By using the pointer to a node in $\boldsymbol{S}_{parent\ (q)}$, the nodes can easily be found. They are: $\{C_1, B_3, A_2\}$, $\{C_1, B_2, A_1\}$, and $\{C_1, B_1, A_1\}$. Nodes on a higher level of a stack cannot be an ancestor of any node on a lower level of the stack. It is because the data nodes are processed in pre-order. In this example, A_2 is not the ancestor of B_2. So $\{C_1, B_2, A_2\}$ is not the query result. The processing time is linear to the size of the data streams and the space needed is $\boldsymbol{O}$ ($d \times |Q|$), where d is the maximal depth of the data set. In this example, it is 6.

In order to do a linear merge in the second phase, a technique was introduced to get all path matches sorted so that higher matching query nodes appear first. In [6], the so-called "self- and inherit-lists" for each stacked node were used to delay out-of-order outputs. Figure 2.2 shows the list for the data and query in Figure 2.1.

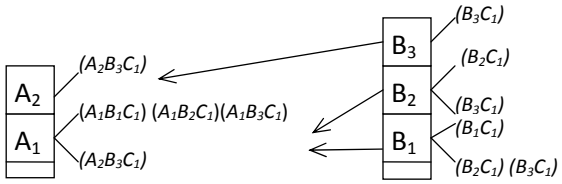


Figure 2.2 A list for the data and query of *TwigStack*

When a node *v* is popped out of a stack, in order to maintain the correct output order, the contents of its lists are appended to the inherit-lists of the node below *v* in the same stack. For the example shown in Figure 2.1, (B_1, C_1)(B_2, C_1)(B_3, C_1) is appended to B_1, (B_2, C_1)(B_3, C_1) to B_2, etc. But if there are some ancestor nodes in the parent stack, the popped node *v* can use,

while the node below *v* in the same stack cannot, decided by the inter-stack pointers, the contents of the lists, appended to its self-list. As shown in the example, popped node B_3 leads to adding (A_2, B_3, C_1) to the self list of A_2.

Algorithm 1 TwigStack

1: **Function** *TWigStack*(Q)
2: **While not** *atEnd*(Q)
3: ***q* : =** *getNext*($Q.root$)
4: **if not** *isRoot*(q)
5: *cleanStack*($S_{parent\,(q)}$, C_q)
6: **if** *isRoot*(q) **or not** *empty*
7: *cleanStack* (S_q, C_q)
8: *push* (S_q, C_q, *top*($S_{parent}(q)$))
9: **if** *isLeaf* (q)
10: *outputPathsDelayed*(C_q)
11: *pop* (S_q)
12: *advance*(T_q)
13: *mergePathSolutions*()

14: **function** *getNext*(q)
15: **if** *isLeaf* (q)
16: **return** q
17: **For** $q_i \in children\,(q)$
18: $q_j = getNext\,(qi)$
19: **if** $q_j \neq q_i$
20: **return** q_j
21: $q_{\min} = \min \arg_{q_i \in children\,(q)} \{c_{q_i}.begin\}$
22: $q_{\max} = \max \arg_{q_i \in children\,(q)} \{c_{q_i}.begin\}$
23: **while** $C_q.end < C_{q_{\max}}.begin$
24: ***advance* (C_q)**
25: **if** $C_q.begine < C_{q_{\min}}.begin$
26: **return** q
27: **else**
28: **return** q_{min}

The pseudo-code for *TwigStack* is shown in ***Algorithm 1*** [4], in which each query node q is associated with a stream T_q and a stack S_q. The current element in T_q is represented by C_q. The recursive function *getnext*(q) is the core of *TwigStack*. It returns a locally highest query node in the subtree of q, and check the heads of the streams of all child query nodes to see if they are all contained by C_q. If it is the case and all child nodes recursively satisfy this requirement, then push C_q into S_q. By this scheme, when a leaf node is pushed into a stack, a path matching is found. But the output is delayed to make sure that the paths are sorted by the top-down order of the query nodes. The *getnext*() function traverses bottom up, and it will jump out if some node does not have a solution extension (see line 20). Leaves don't have any solution extensions. By a recursive execution of *getnext*() function, the query tree is traversed. The return value of each recursive call of *getnext*() is a query node q such that C_q has a descendant of C_{q_i} in T_{q_i} for each child node q_i of q, and each C_{q_i} has recursively the same property as C_q.

In Figure 2.3 shows the process of evaluating the query (b) to data set (a).

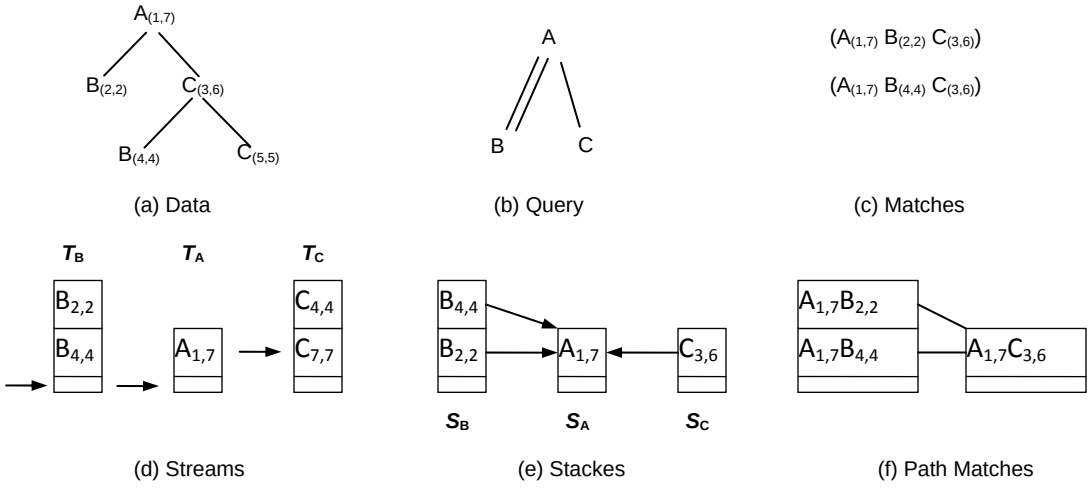


Figure 2.3 *TwigStack* evaluating query

After node $B_{(2, 2)}$ has been processed, for the first call of *getNext*(A), A itself is returned as all the heads of the streams of all child query nodes of A are contained by C_A, and $C_A = A_{(1, 7)}$. For the second call *getNext*(A), $B_{(2, 2)}$ has a usable ancestor $A_{(1, 7)}$ in the parent stack $\boldsymbol{S}_A$, the subtree rooted at $B_{(2, 2)}$ is usable. So $C_B = B_{(2, 2)}$ is pushed into its own stack $\boldsymbol{S}_B$. Since it is a leaf, the path matching $(A_{(1, 7)}, B_{(2, 2)})$ is output. After all paths have been found they are merge joined.

Shortage of *TwigStack*: For mixed A-D and P-C queries, *TwigStack* may perform many redundant checks in calls function *getnext*(). As shown in Figure 2.4, the algorithm cannot always decide whether the data nodes used can satisfy all their P-C relationships by the nodes in the stacks and the heads of the streams. For example, in Figure 2.4, whether the path matches (A_1, B_1), …, (A_1, B_N) are part of a full match or not, cannot be decided before the node C_{N+1} is processed. So, many redundant check is performed. In fact, in the worst case, *TwigStack* needs $\boldsymbol{O}(|D|^{|Q|})$ time for doing the merge joins [7].

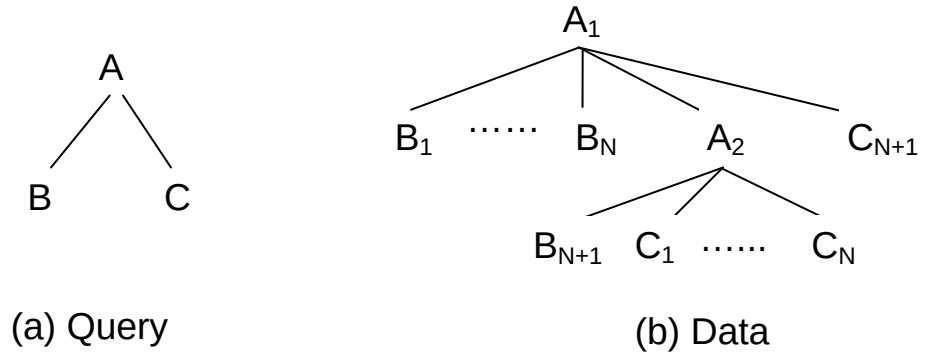


Figure 2.4 Shortage of *TwigStack*

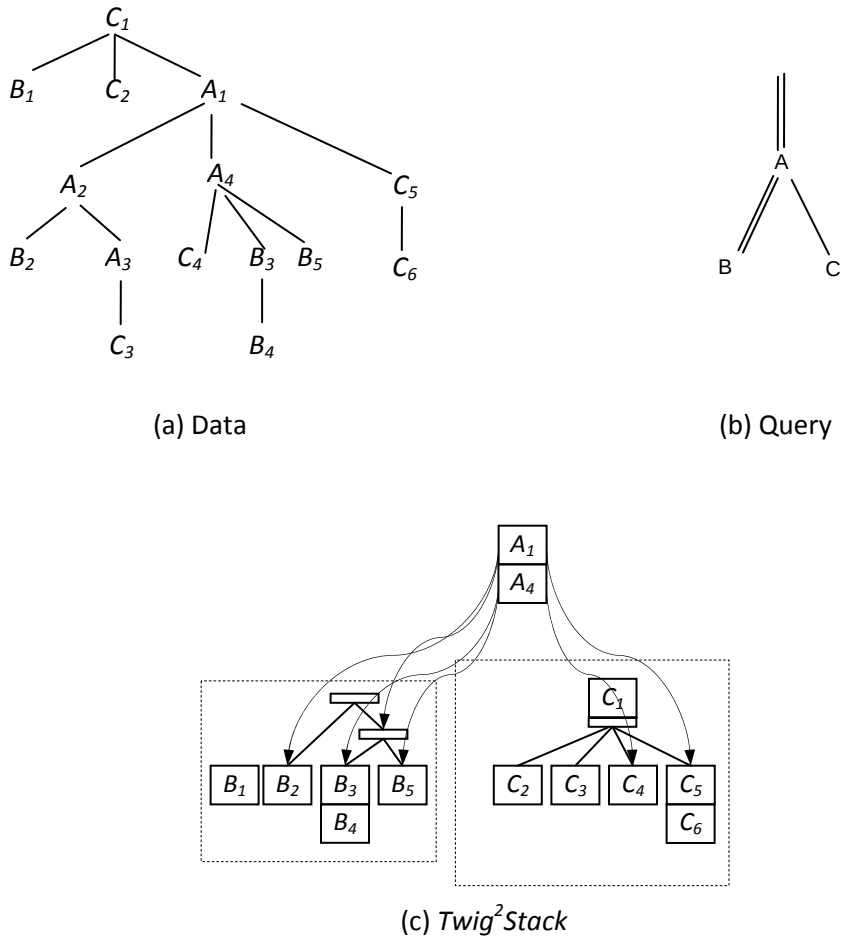
And for A-D only queries, *TwigStack* can solve the problem with a bound of $\boldsymbol{O}(d \times |Q|)$ memory (d is the maximal depth of data set), but for A-D and P-C

mixed queries, it will require $\boldsymbol{O}(n^{min\ n,d}|D|)$ disk space in the worst case[8], where n is the number of structurally recursive labels and *D* is the size of the document.

2.3 Improvements of holistic twig join

A lot of different improvements [7, 10, 16, 17, 18, 19, 20] have been proposed since the introduction of *TwigStack*. We will give a review for some of these algorithms.

Twig²Stack [7] uses the post order sorting for all query nodes. By using a hierarchical stack as shown in Figure 2.5, it can decide whether the entire subtree has a match when the top node is encountered. While processing a query, for each query node, a tree is maintained, in which each node is a stack, as shown in Figure 2.5. In a stack, a data node strictly nests all nodes below and all nodes in the child stacks. The lists of trees are stored in post-order, and are linked together by a common root when an ancestor node is processed. For example, A_1 is linked to B_2, B_3, and B_5. By the post-order, the nodes to be linked will always be found at the end of the list, and the new root will always be put at the end. The order maintains itself naturally. So the nodes' locations will be very clear. Instead of pointing each node in a stack to its ancestor node in its parent stack as in *TwigStack*, *Twig²Stack* points each stacked data node to related child query node. So, a top down list of matching nodes is achieved by this scheme. A node is added only if A-D and P-C relationships can be satisfied, and a P-C pointer is added only when levels are correct, as shown by the P-C pointer: A_1 to C_5 and A_4 to C_4 in Figure 2.5.



(a) Data (b) Query

(c) *Twig²Stack*

Figure 2.5 Hierarchies of stacks for *Twig²Stack*

TwigList [10] is a simplification of *Twig²Stack* using simple lists and intervals given by pointers, which improves performance in practice. For each query node, there is a post-order list of the data nodes encountered so far. As shown in Figure 2.6, by using the same data set as Figure 2.5, each node in a list has, for each child query node, a single recorded interval of contained nodes, such as A_4 contains B_4, B_3, and B_5. Interval start and end positions are recorded as nodes are pushed into and popped out of the global stack. All descendant data nodes are processed in between. Compared with the list of pointers in *Twig²Stack*, the enumeration of matches is not efficient

for P-C edges, but sibling pointers (as B_3 to B_5 showing below) can remedy this.

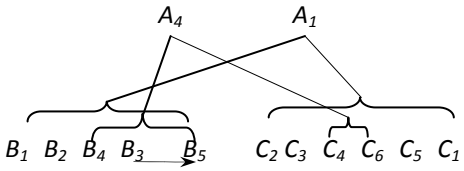


Figure 2.6 Intervals for *TwigList*

HolisticTwigStack [16] modifies *TwigStack* by using a pre-order processing, but maintaining a complex stack structure like *Twig*^{2}*Stack*. The argument against *Twig*^{2}*Stack* is a high memory usage, caused by the fact that all query leaf matches are kept in memory until the tree is completely processed, as they could be part of a match. *HolisticTwigStack* differentiates between the top-most branching node and its ancestors, for which a regular stack is used, and the lower query nodes, which have multiple linked lists of stacks, as shown in Figure 2.7. Each query node match has one pointer to the first descendant in pre-order for each child query node. For "lower" query nodes, new data nodes are pushed into the current stack if contained; otherwise, a new stack is created and appended to the list. As a match for an "upper" query node is popped, the node below it in the corresponding stack must inherit the pointers. For instance, node A_1 would inherit the pointers from both A_2 and A_4 in the example shown in Figure 2.7. Also, the related lists of child matches would be linked.

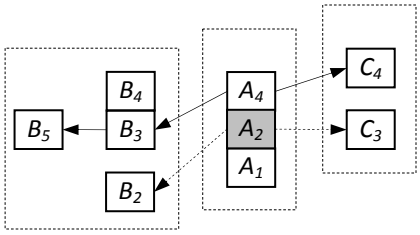


Figure 2.7 Lists of Stacks of *HolisticTwigStack*

TwigFast [1] further simplifies *HolisticTwigStack* and works in a way similar to *TwigList*. There is one list containing matches for each query node, sorted in pre-order. The data nodes in the lists have pointers, giving the interval of the contained matches for each child query node, as shown in Figure 2.8. Each data node put into a list has a pointer to its closest ancestor in the same list, and a "tail pointer", which gives the last position where a node can be the ancestor of the subsequent nodes in the streams. These pointers are used for the construction of the intervals.

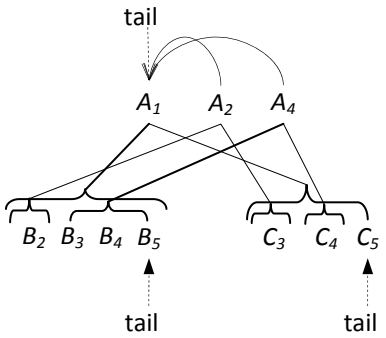


Figure 2.8 Intervals for *TwigFast*

Chapter 3

Twig Pattern Search

This Chapter describes the method that I implemented to evaluate tree pattern queries in a document database. The main purposes of this method are: 1) to efficiently retrieve all matching documents form a database for a give query; 2) to avoid expensive join operations which many index-based methods have to do. To achieve these purposes, we devise a method based on two basic techniques: holistic structure twig join and ***XB-tree*** structure. The holistic structure twig join algorithm treats every document as a set of data streams, and checks the query tree against each document tree to find out whether the query tree can be successfully embedded in it. The ***XB-tree*** technique helps to speed up the process of twig join by: 1) dramatically reducing the number of documents that the tree matching algorithm needs to check; 2) eliminating unnecessary subtree checking. By combining these two powerful techniques, we are able to efficiently find all the documents matching a given query without involving any join operations.

3.1 Tree encoding

An efficient tree encoding scheme was presented in [9]. It can be used to identify different relationships among the nodes of a tree.

Assume T is a document tree as shown in Figure 3.1. We represent each node v in T by a quadruple (*DocId*, *LeftPos*, *RightPos*, *LevelNum*), denoted as $\alpha(v)$, where *DocId* is the document identifier, *LeftPos* and *RightPos* are generated by counting word numbers from the beginning of the document until the start and end of the element, respectively; and LevelNum is the nesting depth of the element in the document. (See Figure 3.1) By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined [9]:

a) *ancestor-descendant*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is an ancestor of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$.

b) *parent-child*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is the parent of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$ and $ln_2 = ln_1 + 1$.

c) *from left to right*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is to the left of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $r_1 < r_2$

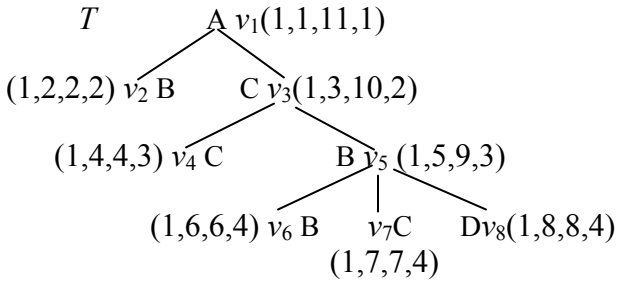


Figure 3.1 Tree Encoding

In Figure 3.1, v_3 is an ancestor of v_7 , as we have $v_{7 \cdot LeftPos} = 3 < v_{6 \cdot LeftPos} = 7$ and $v_{3 \cdot RightPos} = 10 > v_{7 \cdot RightPos} = 7$. Another example, v_5 is the parent of v_7 , as we have $v_{5 \cdot LeftPos} = 5 < v_{6 \cdot LeftPos} = 7$ and $v_{3 \cdot RightPo\ s} = 9 > v_{7 \cdot RightPos} = 7$, as well as

$v_{5 \cdot level} = 3$ and $v_{6 \cdot level} = 4$, which satisfy condition (b). By using the same method as stated above, we can verify all other relationships of the nodes in the tree. In addition, for simplify, if any leaf node v, we set $v_{\cdot LeftPos} = v_{\cdot RightPos}$.

3.2 Main Algorithm

In this section, we discuss our algorithm according to Definition 1 given in section 1.2. The main idea of this algorithm is to reconstruct a sub-tree from the corresponding data streams (a set of quadruple sequences). In the following section, we will separately discuss the subtree reconstruction and twig patterns checking for A-D and P-C relationships in queries.

3.2.1 Tree reconstruction

3.2.1.1 DateStream generation

Using the same notations as [4], we associate each node q in a twig pattern (query tree)Q with a data stream $B(q)$, which contains quadruples (the representation of the node position) of the database nodes v that has the same tag with q. All the quadruples in a data stream are sorted by their (DocID, LeftPos) values. For example, in Figure 3.2, we show a query tree containing 5 nodes and 4 edges and each node is associated with a list matching nodes of the document tree shown in Figure 3.1. For simplicity, the node's name is shown, instead of its quadruple.

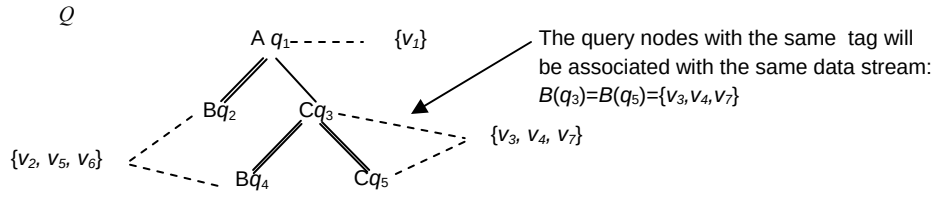


Figure 3.2 Illustration for $B(q_i)$'s

In Q, we can find that multiple query nodes may share the same data stream. So we use gq to represent a group of such query nodes and use $B(gp)$ to denote the data stream shared by them. For example, the nodes in Q shown in Figure 3.2 can be categorized into three groups: $gq_1=\{ q_1\}$, $gq_2=\{ q_2, q_4\}$, and $gq_3=\{ q_3, q_5\}$. Then, $B(gq_1)=\{v_1\}$, $B(gq_2)=\{v_2, v_5, v_6\}$, and $B(gq_3)=\{v_3, v_4, v_7\}$.

3.2.1.2 DataStream Transition

The nodes in each data stream are sorted by their *LeftPos* values, as the access of document nodes is done in preorder. However our algorithm needs to visit them in postorder (in sorted order of their *RightPos* values). For this reason, we designed a global stack *ST* to make a transformation of data streams. The detail process is shown in ***Algorithm* 2**. In *ST* each entry is a pair (gq, v) with $gq \subseteq Q$ and $v \in T$.

Algorithm 2 stream-transformation($B(gq_i)$'s)

input: all data streams $B(gq_i)$, each sorted
output: new data streams $L(gq_i)$, each sorted by RightPos

begin

1: repeat until each $B(gq_i)$ becomes empty
2: { identify gq_i such that the first element v of $B(gq_i)$ is of
3: the minimal LeftPos value;
4: **while** *ST* is not empty and *ST.top* is not *v*'s
5: {$x \leftarrow ST.\ pop(\ )$; Let $x = (gq_i, u)$; $top(S_{parent}(q))$)
6: put u at the end of L(gq_i); }
7: $ST.push(q_i, v)$;
8: }

end

In this algorithm, *ST* is used to maintain all the nodes on a path of in a document tree until we meet a node *v* which is not a descendant of *S.top*() (see line 2 &3). Then, we pop out all those nodes which are not an ancestor of *v*, and then push *v* into *ST* (see lines 4 - 5). The output of the algorithm is a set of data streams $L(gq_i)$'s and the nodes in it are all sorted by *RightPos*. Since the popped nodes themselves are listed in postorder (see line 3), so we can directly process them in postorder without explicitly generating $L(gq_i)$'s. Just for ease explanation, we will assume $L(gq_i)$'s are completely generated in the following discussion. So We use *gq* to represent a set of such query nodes and denote, by $L(gq)$, the data stream shared by them. We also assume that the query nodes in *gq* are sorted by their *RightPos* values. Furthermore, we will use $L(Q)=\{ L(gq_1), \ldots, L(gq_n)\}$ to represent all the data streams with

respect to Q, where each q_i (i=1, …, n) is a set of sorted query nodes which share the same data stream.

3.2.1.3 Reconstruction

Before we discuss how to reconstruct a tree structure from the data streams, we would like to introduce another conception of *matching subtrees*. Denote a tree by T and v is a node in T which has a parent node u. Denote another tree by T' which is obtained by removing node v. This process is denoted by *delete*(T, v) and the children of v_3 become the children of v_1 (see Figure. 3.3)

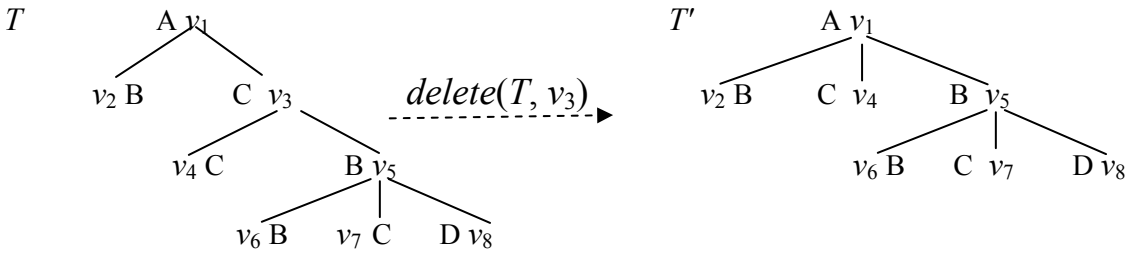


Figure 3.3 T' is achieved by removing v_3 from T

Definition 2.(*matching subtree*) A matching subtree T' of T with respect to a twig pattern Q is a tree obtained by a series of deleting operations to remove any node in T, which does not match any node in Q.

According to this definition, the tree shown in Figure 3.4(a) with respect to the query tree shown in Figure 3.4(b) is a matching subtree which is obtained by a series of node deleting (in this case the nodes contain tag C are deleted) from the document tree shown in Figure 3.1.

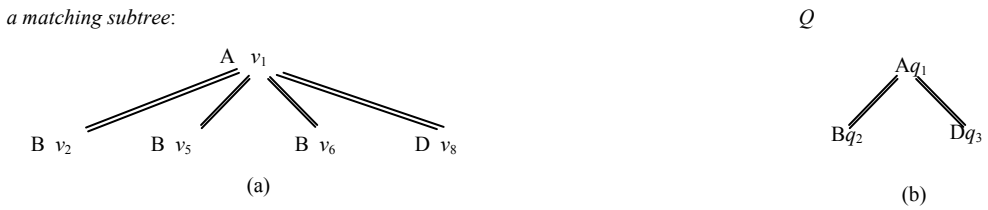


Figure 3.4. Matching tree obtained based on a query tree

Based on this *matching subtrees*, we can design a recursive process to access the nodes in $L(gq_i)$'s one by one, and a subtree structure T' of T can be constructed as below:

1. Identify a data stream $L(q)$ with the first element being of the minimal *RightPos* value. Choose the first element v of $L(q)$. Remove v from $L(q)$;

2. For each popped node, generate a node for v;

3. If v is not the first node created, let v' be the node chosen just before v, and do the following two steps.

   a) If v' is not a child or descendant of v, create a link from v to v', called a *left-sibling link* and denoted as *left-sibling*(v) = v'.

   b) If v' is a child or descendant of v, we will first create a link from v' to v, called a *parent link* and denoted as *parent*(v') = v. Then, we will go along the left-sibling chain starting from v' until we meet a node v'' which is not a child or a descendant of v. For each encountered node u except v'', set *parent*(u) ← v. Finally, set *left-sibling*(v) ← v''.

Construction process is shown in Figure 3.5.

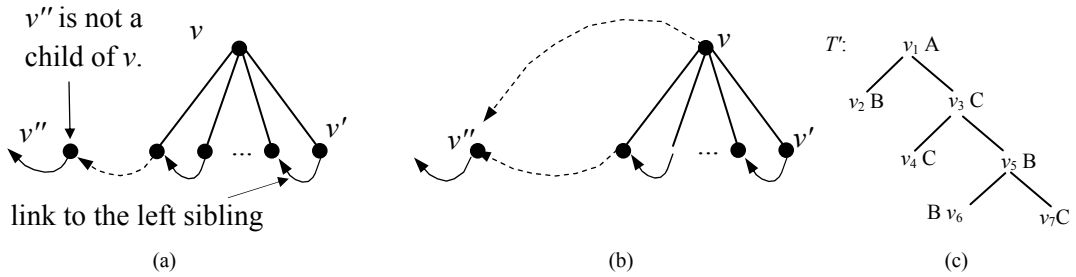


Figure 3.5. Illustration for the construction of a matching subtree

In Figure 3.5(a), you will find that v' is a child (descendant) of v. If this condition is satisfied, the navigation goes along a left-sibling chain starting from v' to the next sibling until meet v'', a node that is not a child (descendant) of v. In Figure 3.5 (b) a left-sibling link of v is set to v'', which is previously navigated from the left-sibling link of v's leftmost child. Applying the above process to $B(\boldsymbol{q}_i)'$s shown in Figure. 3.2, we will regain a tree T', called a *matching subtree*, as shown in Fig. 3.5(c). This is similar to the tree shown in Figure 3.3, but with node v_3 being removed.

The Algorithm 3 gives the detail of this reconstruction process, as shown in next page.

Algorithm 3 ***subtree reconstruction***

input: all data streams $L(Q)$.
output: a matching subtree

begin

1: **repeat until** each $L(gq)$ in $L(Q)$ become empty
2: {identify gq such that the first element v of $L(gq)$ is of the minimal RightPos value; remove v from $L(q)$;
3: generate node v;
4: if v is not the first node created **then**
5: { let v' be the node genertated just before v;
6: **if** v' is not a child (descendant) of v **then**
7: { *left-sibling*(v) ← v';} (*generate a left-sibling link*)
8: **else**
9: { $v'' \leftarrow v'$; $w \leftarrow v'$;} (*v'' and w are two temporary variables.*)
10: **while** v'' is a child (descendant) of v **do**
11: { *parent* (v'') ← v; (* generate a parent link. Also, indicate whether v'' is a /-child or a //-child. *)
12: $w \leftarrow v''$; $v'' \leftarrow$ *left-sibling*(v'');
13: }
14: *left-sibling* (v) ← v'';
15: }
16: }

end

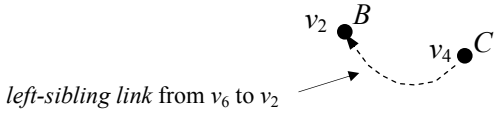
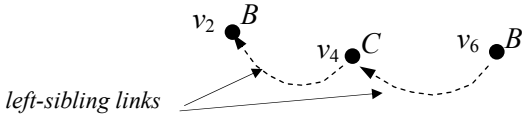
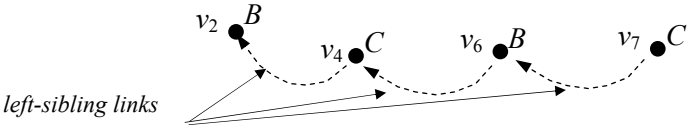
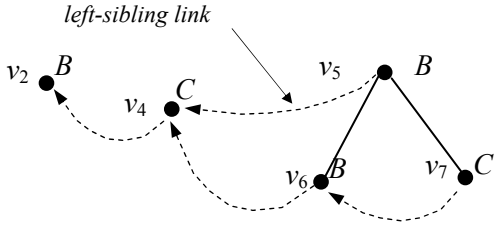
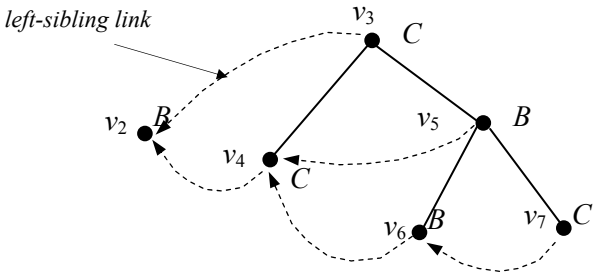
In the above algorithm, a new node is created for each chosen v from a $L(gq)$. Assume that v' has already created before v, if v' is not a child or descendant of v (see line 7), create a left-sibling link from v, pointing to the node v'. Otherwise, we need to go into a **while**-loop (see line 10) to travel along the left-sibling linked list starting from v' until we meet a node v'' which is not a child or descendant of v. During this process, a parent link is generated for

each node encountered except v''. (see lines 9-13). Finally, the left-sibling link of v is set to v'' (see line 14).

In order to make a more brief explanation, we make an illustration of this data tree generating process, shown as Figure 3.6(based on the data tree shown in Figure 3.1 and query tree shown in Figure 3.2).

data stream: $L(gq_1)= \{v_1\}$, $L(gq_2)= \{v_2, v_6, v_5\}$, $L(gq_3)= \{v_4, v_7, v_3\}$

$gq_1 = \{q_1\}$, $gq_1 = \{q_2, q_4\}$, $gq_1 = \{q_3, q_5\}$

	v with the least *RightPos*:	generated data structure:
Step 1:	v_2	
Step 2:	v_4	
Step 3:	v_6	
Step 4:	v_7	
Step 5:	v_5	
Step 6:	v_3	

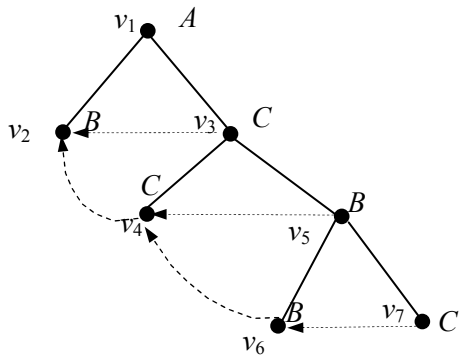
	v with the least *RightPos*:	generated data structure:
Step 7:	v_1	

Figure 3.6 Sample trace for ***Algorithm 3***

In step 1(see Figure 3.6), v_2 is checked since it has the least RightPos value; and a node for it is created. In Step 2, we meet v_4. Since v_2 is not a descendant of v_4, we establish a left-sibling link from v_4 to v_2. In Step 3, we meet v_6. For the same reason as Step 2, we establish a left sibling link from v_6 to v_4. In step 4, we establish a left sibling link from v_7 to v_6. In step 5 we meet v_5. Since v_7 is the child of v_5, we generate an edge between them, and then navigated to v_6, which is also a child of v_5. So, an edge from v_5 to v_6 is generated. In this step, not only two edges are constructed, but also a left-sibling link from v_5 to v_4 is generated. This is the key link that enables us to reconstruct a matching subtree in an efficient way. The following steps are shown in Figure 3.6 above.

3.2.1.4 Correctness of Algorithm 3

In this section, we will prove the correctness of the algorithm *matching-tree-reconstruction*.

Proposition 1 Denote a document tree as T and a twig pattern as Q. Let $L(Q)= \{L(gq_1), \ldots, L(gq_n)\}$ be all the data streams based on Q and T, where each q_i $(1 \le i \le n)$ is a subset of Q, in which query nodes is sorted, and share the same data stream. Algorithm *matching-tree-construction* generates the matching subtree T' of T with respect to Q correctly.

Proof. Denote $L = |L(gq_1)| + \ldots + |L(gq_n)|$. We prove the proposition by induction on L.

Basis. When L=1, the proposition holds.

Induction hypothesis. Assume that when $L = k$, the proposition holds.

Induction step.

We consider the case when $L = k + 1$. Assume that all the quadruples in $L(Q)$ are $\{u_1, \ldots, u_k, u_{k+1}\}$ with $RightPos\ (u_1) < RightPos\ (u_2) < \ldots < RightPos\ (u_k) < RightPos\ (u_{k+1})$. The algorithm will first generate a tree structure T_k for $\{u_1, \ldots, u_k\}$. In terms of the induction hypothesis, T_k is correctly created. It can be a tree or a forest. If it is a forest, all the roots of the subtrees in T_k are connected through left-sibling links. When we meet v_{k+1}, we consider two cases:

a) v_{k+1} is an ancestor of v_k;

b) v_{k+1} is at the right of v_k.

In case a), the algorithm will generate an edge (v_{k+1}, v_k), and then travel along a left-sibling chain starting from v_k until we meet a node v which is not a descendant of v_{k+1}. For each node v' encountered, except v, an edge (v_{k+1}, v') will be generated. Therefore, T_{k+1} is correctly constructed. In case b), the

algorithm will generate a left-sibling link from v_{k+1} to v_k. It is obviously correct since in this case v_{k+1} cannot be an ancestor of any other nodes. The proof completes.

The time complexity of this process is easy to analyze. First, we notice that each quadruple in all the data streams is accessed only once. Secondly, for each node in T', all its child nodes will be visited along a left-sibling chain for a second time. So we get the total time

$$O(|D| \cdot |Q|) + \sum_i d_i = O(|D| \cdot |Q|) + O(|T'|) = O(|D| \cdot |Q|)$$

where d_i represents the outdegree of node v_i in T'.

During the process, for each encountered quadruple, a node v will be generated. Associated with this node have we at most two links (a left-sibling link and a parent link). So the used extra space is bounded by $\boldsymbol{O}(|T'|)$.

3.2.2 Tree Matching

In fact, ***Algorithm 3*** hints an efficient way for twig pattern matching.

We observe that during the reconstruction process of a matching subtree T'. we can also associate each node v in T' with a query node stream $QS(v)$. That is, each time we choose a v with the largest *LeftPos* value from a data stream $L(gq)$, we will insert all the query nodes in gq into $QS(v)$. For example, in the first step shown in Figure 3.6, the query node stream for v_2 can be determined as shown in Figure 3.7(a).

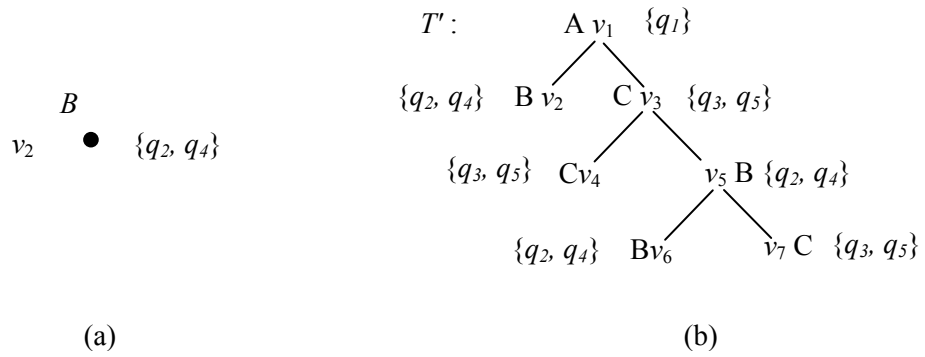


Figure 3.7 Illustration of generating *QS*′s

In the same way, we can create the whole matching subtree as shown in Figure 3.7(b), each node in T' is associated with a sorted query node stream. If we check, before a q is inserted into the corresponding $QS(v)$, whether $Q[q]$ (subtree rooted at q) can be imbedded into $T'[v]$ or not, we actually get an algorithm for twig pattern matching. The only problem left is how to make an efficient checking.

For this purpose we can associate each q in Q with a variable, denoted $\gamma(q)$. During the process, $\gamma(q)$ will be dynamically assigned a series of values a_0, a_1, …, a_m for some m in sequence, where $a_0 = \Phi$ and a_i's ($i = 1, \ldots, m$) are different nodes from T'. In another word, we just use these symbols to represent a specific node in T'. Initially, $\gamma(q)$ is set to $a_0 = \Phi$. $\gamma(q)$ will be changed from a_{i-1} to $a_i = v$ ($i = 1, \ldots, m$) while the following conditions are satisfied.

i) v is the node currently encountered.

ii) q appears in $QS(u)$ for some child node u of v.

iii) q is a //-child,

or q is a /-child, and u is a /-child with label(u) = label(q).

Then, each time before we insert q into $QS(v)$, we will do the following checking:

1. Let $q_1, \ldots, q_k$ be the child nodes of q.
2. If for each q_i ($i = 1, \ldots, k$), $\gamma(q)$ is equal to v and label(v) = label(q), insert q into $QS(v)$.

As the matching subtree is constructed in a bottom-up way, the above checking is guaranteed that for any $q \in QS(v)$, $T'[v]$ contains $Q[q]$.

Let $v_1, \ldots, v_j$ be the children of v in T'. All the QS(v_i)'s ($i = 1, \ldots, j$) should also be added into $QS(v)$. This process can be elaborated as follow:

Let $QS(v_i) = \{ q_{i_1}, \ldots, q_{i_j} \}$ ($i = 1, \ldots, j$).

Pay attention to the complex symbol $\{ q_{i_1}, \ldots, q_{i_j} \}$ here, it means, for example, if i=1, it means that v has only one child and $QS(v_1)$ will be $\{ q_{i_1}, \ldots, q_{i_j} \}$ $\{i = 1, \ldots, j)$, in which the query nodes sharing the same tag is $\{ q_{i_1}, \ldots, q_{i_j} \}$ $\{i = 1, \ldots, j)$.

Then, we have $q_{i_1}.LeftPos < \ldots < q_{i_j}.LeftPos$. Because, all the query nodes inserted into $QS(v_i)$ come from a same set "gq", in which all the elements are sorted by their *LeftPos* values. Each time we insert a q into $QS(v_i)$, we can check whether it is subsumed by the query node q' which has just been inserted before. If it is subsumed by the node q' which is inserted before, q will not be inserted, since the embedding of $Q[q']$ in $T[v_i]$ implies the embedding of $Q[q]$ in $T[v_i]$ (As the reason that $LeftPos(q') < LeftPos(q)$, q

cannot be an ancestor of q'.) Thus, $QS(v_i)$ contains only the query nodes which are on different path. Therefore, we must also have $q_{i_j}.RightPos < \ldots < q_{i_j}.RightPos$ (As the reason that $LeftPos(q') < LeftPos(q)$, if $RightPos(q') > RightPos(q)$, q' will be the ancestor of q, so $Q[q']$ in $T[v_i]$ implies the embedding of $Q[q]$ in $T[v_i]$, which not satisfy the condition we discussed above just now). So the query nodes in $QS(v_i)$ are increasingly sorted by both *LeftPos* and *RightPos* values. Obviously, $|QS(v_i)| \leq Leaf_Q$(all the leaf node in Q). We can store $QS(v_i)$ as a linked list. Let QS_1 and QS_2 be two sorted lists with $|QS_1| \leq leaf_Q$ and $|QS_2| \leq leaf_Q$. The union of QS_1 and QS_2 ($QS_1 \cup QS_2$) can be performed by scanning both QS_1 and QS_2 from left to right and inserting the query node of QS_2 into QS_1 one by one. During this process, any query node in QS_1, which is subsumed by some query node in QS_2 will be removed; and any query node in QS_2, which is subsumed by some query in QS_1, will not be inserted into QS_1, The result is stored in QS_1. From this , we can see that the resulting linked list is still sorted and its size is bounded by $leaf_Q$. We denote this process as $merge(QS_1, QS_2)$ and define $merge(QS_1, \ldots, QS_{j-1}, QS_j)$ to be $merge(merge(QS_1, \ldots, QS_{j-1}), QS_j)$, it's a recursive way.

In the following, we will present ***Algorithm 4***. *twig pattern matching*, which is an enhance of ***Algorithm 3***. The main idea can be simply described as follow: While we are constructing the matching subtree T' of T as ***Algorithms 3***, we append only the "correct related" nodes into an QS, ("correct related" means the nodes which satisfy the 3 conditions we discussed above), and store them in a linked list $QS(v)$, then the twig matching result can be generated with the T' reconstruction process automatically.

Algorithm 4 ***twig pattern matching***

input: all data streams $L(Q)$.
output: a matching subtree T' of T, represented by a data stream $QS(v)$

begin

1: **repeat until** each $L(gq)$ in $L(Q)$ become empty
2: {identify gq such that the first element v of $L(gq)$ is of the minimal RightPos value; remove v from $L(q)$;
3: generate node v;
4: **if** v is not the first node created **then**
5: { $QS(v) \leftarrow$ *subsumption-check*(v,q);}
6: **else**
7: {let v' be the quadruple chosen just before v, for which a node is constructed
8: **if** v' is not a child (descendant) of v **then**
9: { *left-sibling*$(v) \leftarrow v'$; $QS(v) \leftarrow$ *subsumption-check*(v,q);}
10: **else**
11: { $v'' \leftarrow v'$; $w \leftarrow v'$;} (*v'' and w are two temporary variables.*)
12: **while** v'' is a child (descendant) of v **do**
13: { *parent* $(v'') \leftarrow v$; (* generate a parent link. Also, indicate whether v'' is a /-child or a //-child. *)
14: **for** each q in $QS(v'')$ **do** {
15: **if**((q is a //-child) or (q is a /-chld and v'' is a /-child and *label*(q) = *label*(v'')))
16: **then** $\gamma(q) \leftarrow v$;}
17: $w \leftarrow v''$; $v'' \leftarrow$ *left-sibling*(v'');
18: remove *left-sibling*(w);
19: }
20: $gq \leftarrow$ *subsumption-check*(v,gq);
21: let $v_1, \ldots, v_j$ be the child nodes of v;
22: $gq' \leftarrow$ *merge*$(QS(v_1), \ldots, QS(v_j))$;
23: remove $QS(v_1), \ldots, QS(v_j)$;
24: $QS(v) \leftarrow$ *merge*(gq, gq') ;}}

end

Fuction *subsumption-check*(*v*,*gq*) (**v* satisfies the node name test at each *q* in *gq*.*)

begin

1: {$QS \leftarrow \Phi$;

2: **for** each *q* in *gq* **do**

3: {let $q_1, \ldots, q_j$ be the child nodes of *q*;

3: generate node *v*;

4: **if** for each /-child q_i $\gamma(q_i) = v$ and for each //-child q_i $\gamma(q_i)$ is subsumed by *v* **then**

5: {

6: $QS \leftarrow QS \cup \{q\}$;

7: }

8: return *QS*;

9: }

end

Algorithm 4 does almost the same work as ***Algorithm 3*** *matching-tree-reconstruction*(). The main difference is lines 14 - 18 and lines 20 - 24. In lines 14 - 18, we set γ values for some *q's*. Each of them appears in a $QS(v')$, where v' is a child node of *v*, satisfying the conditions i) ii) iii) given above. In lines 20 - 24, we use the merging operation to construct $QS(v)$.

In Function *subsumption-check*(), we check whether any *q* in *gq* can be inserted into *QS* by examining the A-D and P-C relationships between nodes (see line 4). Continuing this process, we can find that T' embeds *Q*.

Applying ***Algorithm 4*** to the data set shown in Figure 3.1, we will find the document tree shown in Figure 3.1, contains the query tree shown in Figure 3.2. We trace the computation process as shown in Figure 3.8.

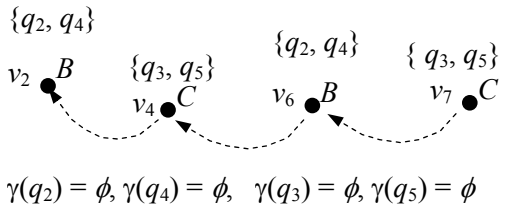
data stream: $L(gq_1)= \{v_1\}, L(gq_2)= \{v_2, v_6, v_5\}, L(gq_3)= \{v_4, v_7, v_3\}$

$gq_1 = \{q_1\}$, $gq_1 = \{q_2, q_4\}$, $gq_1 = \{q_3, q_5\}$

For simplicity, we start at step 4

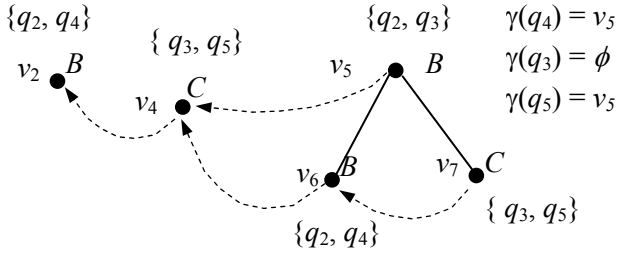
	v with the least RightPos:	generated data structure:

Step 4: v_7



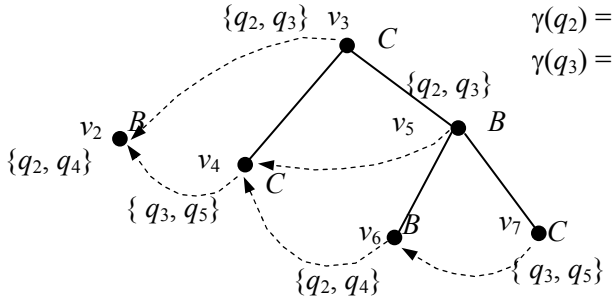
$\gamma(q_2) = \phi, \gamma(q_4) = \phi, \quad \gamma(q_3) = \phi, \gamma(q_5) = \phi$

Step 5: v_5



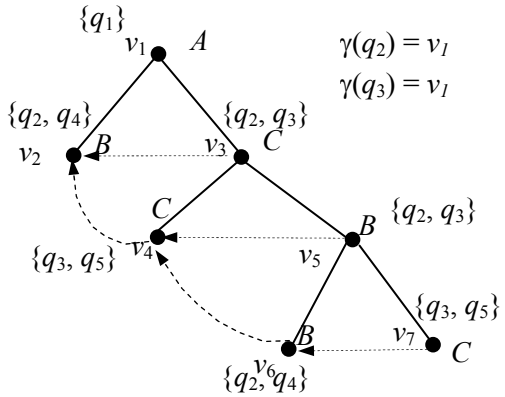
$\gamma(q_2) = v_5$
$\gamma(q_4) = v_5$
$\gamma(q_3) = \phi$
$\gamma(q_5) = v_5$

Step 6: v_3



$\gamma(q_2) = v_3$
$\gamma(q_3) = v_3$

Step 7: v_1



$\gamma(q_2) = v_1$
$\gamma(q_3) = v_1$

Figure 3.8 Sample trace for ***Algorithm 4***

In the first four steps, we will generate part of the matching subtree as shown in Figure 3.8 step 4. At this time point, we meet v_7, associate with v_4 and v_7 is a query node stream: $QS(v_4) = QS(v_7) = \{q_3, q_5\}$, $QS(v_2) = QS(v_6) = \{q_2, q_4\}$, and $\gamma(q_2) = \gamma(q_4) = \gamma(q_3) = \gamma(q_5) = \phi$. Because, in the last several steps, we didn't meet any node which is the ancestor/parent of the current nodes (see line 12). In step 5, we meet v_5 (associate with $L(gq)$, $\{q_2, q_4\}$), the parent of v_6 and v_7. Basic on the ***Algorithm 4***, we will check $QS(v_6)$ and $QS(v_7)$ (see line 15, 16), as the reason all the q nodes are //-child except q_3, q_3 is a /-child but v_7 is not a /-child of v_5, so we will get $\gamma(q_2) = \gamma(q_4) = \gamma(q_5) = v_5$, $\gamma(q_3) = \phi$. So q_3 will not satisfy the *subsumption-check*, $QS(v_5) = \{q_5\}$. Then we will merge $QS(v_6)$ and $QS(v_7)$ into $QS(v_5)$, q_4 and q_5 will be subsumed by q_3, so at last $QS(v_5) = \{q_2, q_3\}$. In step 6, we meet v_3 (associated with L(gq), $\{q_3, q_5\}$), the parent of v_4 and v_5, $QS(v_4) = \{q_3, q_5\}$, $QS(v_5) = \{q_2, q_3\}$, as the reason q_2, q_5 is //-child, $\gamma(q_2) = \gamma(q_5) = v_3$. For q_3, it is /-child and the $v''(v_5)$ is a //-child, so $\gamma(q_3) = \phi$. After *subsumption-check* and merging of $QS(v_4)$ and $QS(v_5)$, $QS(v_3) = \{q_2, q_3\}$. In the last step, we meet v_1, according to $QS(v_2) = \{q_2, q_4\}$, $QS(v_3) = \{q_2, q_3\}$, we will set $\gamma(q_2) = \gamma(q_4) = v_1$, as q_2 and q_4 are //-child, q_3 is a /-child, $v''(v_3)$ is also a /-child, and $label(v_3) = \text{label}(q_3)$. So $\gamma(q_3) = v_1$, leading to the insertion of q_1 into $QS(v_1)$. After merging $QS(v_1) = \{q_1\}$. Finally, the embedding Q in T has been generated while the process of constructing T', which mean the twig searching has been finished.

In the following, we will prove the correctness of this algorithm. First, we need to prove a simple lemma.

Lemma 1 Assume v_1, v_2 and v_3 are three nodes in a tree and v_3.LeftPos < v_2.LeftPos < v_1. *LeftPos*. If v_1 is a descendant of v_3. Then, v_2 must also be a descendant of v_3.

Proof. Considering two cases: *i*) v_2 is at the left position of v_1, *ii*) v_2 is an ancestor of v_1. In case *i*), we have v_1.RightPos > v_2.RightPos. So we have v_3.RightPos > v_1.RrightPos > v_2.RrightPos. This shows that v_2 is a descendant of v_3. In case *ii*), v_1, v_2 and v_3 are on the same path. Since v_2.LeftPos > v_3.LeftPos, v_2 must be a descendant of v_3.

Proposition 2 Let Q be a twig pattern containing only /-edges, //-edges and branches. Let v be a node in the matching subtree T' with respect to Q created by Algorithm 4. Let q be a node in Q. Then q appears in $QS(v)$ if and only if $T'[v]$ contains $Q[q]$.

Proof. If-part. A query node q is inserted into $QS(v)$ by executing Function *subsumption-check*(), which shows that for any q inserted into $QS(v)$ we must have $T''[v]$ containing $Q[q]$ for the following reason:

(1) $label(v) = label(q)$.

(2) For each //-child q' of q there exists a child v' of v such that $T[v']$ contains $Q[q']$. (See line 15 in Algorithm 4)

(3) For each /-child q'' of q there exists a /-child v'' of v such that $T[v'']$ contains $Q[q'']$ and $label(v'') = label(q'')$. (see lines 15 in algorithm 4)

In addition, a query node q in $QS(v)$ may comes from a QS of some child nodes of v. Obviously, we have $T'[v]$ containing $Q[q]$.

Only-if-part. The proof of this part is tedious. In the following, we give only a proof for the simple case that Q contains no /-edges, which is done by induction of the height h of the nodes in T'.

Basis. When $h = 0$, for the leaf nodes of T', the proposition holds.

Induction step. Assume that the proposition holds for all the nodes at height $h \leq k$. Consider the nodes v at height $h = k + 1$. Assume that there exists a q in Q such that $T'[v]$ contains $Q[q]$ but q does not appear in $QS(v)$. Then there must be a child node q_i of q such that (*i*) $\gamma(q_i) = \phi$, or (*ii*) $\gamma(q_i)$ is not subsumed by v when q is checked against v. Obviously, case (*i*) is not possible since $T'[v]$ contains $Q[q]$ and q_i must be contained in a subtree rooted at a node v' which is a child (descendant) of v. So $\gamma(q_i)$ will be changed to a value not equal to ϕ in terms of the induction hypothesis. Now we show that case (*ii*) is not possible, either. First, we note that during the whole process, $\gamma(q_i)$ may be changed several times since it may appear in more than one *QS's*. Assume that there exist a sequence of nodes $v_1, \ldots, v_k$ for some $k \geq 1$ with $v_1.LeftPos > v_2.LeftPos > \ldots > v_k.LeftPos$ such that q_i appears in $QS(v_1), \ldots, QS(v_k)$. In terms of the induction hypothesis, $v' = v_j$ for some $j \in \{1, \ldots, k\}$. Let l be the largest integer $\leq k$ such that $v_l.LeftPos > v.LeftPos$. Then, for each v_p ($j \leq p \leq l$), we have

$$v'.LeftPos \geq v_l.LeftPos > v.LeftPos$$

In terms of **Lemma 1**, each v_p ($j \leq p \leq l$) is subsumed by v. When we check q against v, the actual value of $\gamma(q_i)$ is the node name for some v_p*'s* parent, which is also subsumed by v (in terms of **Lemma 1**), contradicting (*ii*). The above explanation shows that case (*ii*) is impossible. The proof of the proposition completes.

Lemma 1 helps to clarify the *only-if* part of the above proof. In fact, it reveals an important property of the tree encoding, which enables us to save both space and time. That is, it is not necessary for us to keep all the values of $\gamma(q_i)$, but only one to check the A-D/P-C relationship. Due to this property, the path join [4], as well as the result enumeration [7], can be completely avoided.

The time complexity of the algorithm can be divided into three parts:

1. The first part is the time spent on accessing $L(Q)$. Since each element in a $L(Q)$ is visited only once, this part of cost is bounded by $\boldsymbol{O}(|D|\cdot|Q|)$

2. The second part is the time used for constructing $QS(v_j)$'s. For each node v_j in the matching subtree, we need $O(\sum_i c_{j_i})$ time to do the task, where c_{j_i} is the outdegree of q_{j_i}, which matches v_j. (See line 2 and 3 in Function *subsumption-check*() for explanation.) So this part of cost is bounded by

$$O(\sum_j \sum_i c_{j_i}) \le O(|D| \cdot \sum_k^{|Q|} c_k) = O(|D| \cdot |Q|).$$

3. The third part is the time for establish γ values, which is the same as the second part since for each q in a $QS(v)$ its γ value is assigned only once.

Therefore, the total time is $\boldsymbol{O}(|D|\cdot|Q|)$.

The space overhead of the algorithm is easy to analyze. Besides the data streams, each node in the matching subtree needs a parent link and a values.

right-sibling link to facilitate the subtree reconstruction, and an QS to calculate γ values. So the extra space requirement is bounded by

$$\boldsymbol{O}(|D|\cdot|Q| + |D| + |Q|) = O(|D|\cdot|Q|).$$

However, if we record only those parts of T', which contain the whole Q or the subtree rooted at the output node, the runtime memory usage must be much less than $O(|D|\cdot|Q|)$ for the following two reasons:

(*i*) The *QS* data structure for a node is removed once its parent node is created. So the space overhead is bounded by $\boldsymbol{O}(|D|\cdot Leaf_Q)$.

(ii) During the whole process, the elements in the data streams are removed one by one.

Of course, if we want to record all those parts of T', which contain one or more parts of Q, we need $O(|D|\cdot|Q|)$ space to store all the results.

In the above discussion, we handle wildcards in the same way as any non-wildcard nodes. But a wildcard matches any tag name. Therefore. $L(*)$ should contain all the nodes in t. However, as we can see in the next section, by using the ***XB*-tree**[4], $L(*)$ contains a much smaller set of nodes in T. In fact, during the whole process each entry in an ***XB*-tree** is accessed only one along the nodes' postorder numbers. That is, for each node in Q, no matter whether it is a wildcard or not, we only check it against the nodes currently encountered. Thus with the help of ***XB*-trees**, * can be handled in the same as non-wildcard, causing no extra time complexity.

3.3 *XB-tree* index

In this section, we discuss how the algorithm presented in the previous section can be adapted to an indexing environment by constructing ***XB-tree*** [4] over data streams.

For each data stream $B(q)$ associated with a certain q, we can establish an ***XB-tree*** [4], which can be considered as a variant of B^+-tree. In such an index structure, each entry in a page is a pair a = (*LeftPos*, *RightPos*) (referred to as a bounding segment) such that any entry appearing in the subtree pointed to by the pointer associated with a is subsumed by a. In addition, all the entries in a page are sorted by their *LeftPos* values. As an example, consider a sorted quadruple sequence shown in Figure 3.9(a), for which we may generate an ***XB-tree*** as shown in Figure 3.9(b).

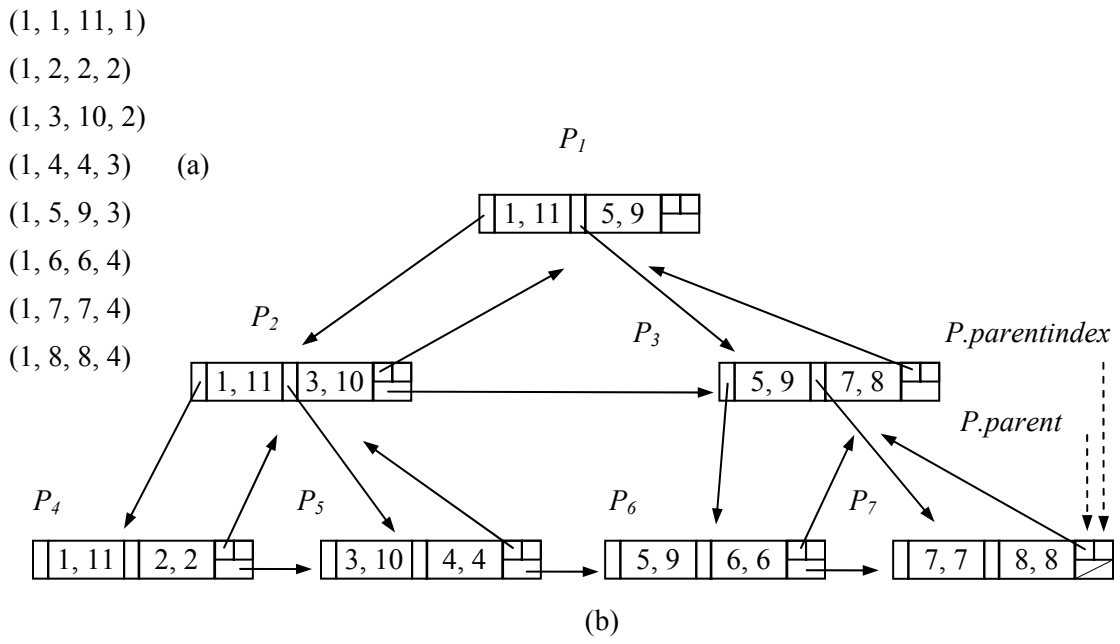


Figure 3.9 Sample of ***XB-tree***

In each page P of an ***XB-tree***, the bounding segments may partially overlap, but their *LeftPos* positions are in increasing order. Besides, it has two extra

data fields: *P.parent* and *P.parentindex* is a number *i* to indicate that the *i*th pointer in *P.parent* points to *P*. For instance, in the ***XB-tree*** shown in Figure 13(b), $P_7.parentIndex = 2$ since the second pointer in P_3 (the parent of P_7) points to P_7.

We notice that in a Q we may have more than one query nodes $q_1, .., q_k$ with the same label. So they will share the same data stream and the same ***XB-tree***. For each q_j ($j = 1, \ldots, k$), we maintain a pair (P, i), denoted σ_{q_j}, to indicate that the ith entry in the page P is currently accessed for q_j. Thus, each σ_{q_j} ($j = 1, \ldots, k$) corresponds to a different searching of the same ***XB-tree*** as if we have a separate copy of that ***XB-tree*** over $B(q_j)$.

In [4], two operations are defined to navigate an ***XB-tree***, which change the value of σ_q

1. *advance*(σ_q)(going up from a page to its parent): if $\sigma_q = (P, i)$ does not point to the last entry of P, $i \leftarrow i + 1$. Otherwise, $\sigma_q \leftarrow (P_{.parent}, P_{.parentIndex} + 1)$.

2. *drilldown*(σ_q) (going down from a page to one of its children): If $\sigma_q = (P, i)$ and P is not a leaf page, $\sigma_q \leftarrow (P', 1)$, where P' is the ith child page of P.

Initially, for each q, σ_q points to (*rootPage*, 0), the first entry in the root page. We finish a traversal of the ***XB-tree*** for q when σ_q = (*rootPage*, *last*), where last points to the last entry in the root page, and we advance it (in this case, we set σ_q to ϕ, showing that the ***XB-tree*** over $B(q)$ is exhausted.) As with ***TwigStackXB***, the entries in $B(q)$'s will be taken form the corresponding ***XB-tree***; and many entries can be possibly skipped. Again, the entries taken from ***XB-tree*** will be reordered as shown in Algorithms 2. *stream-*

transformation(). According to [4], each time we determine a q ($\in Q$), for which an entry from $B(q)$ is taken, the following three conditions are satisfied:

i) For q, there exists an entry v_q in $B(q)$such that it has a descendant v_{q_i} in each of the streams $B(q_i)$ (where q_i is a child of q.)

ii) Each v_{q_i} recursively satisfies (i).

iii) LeftPos(v_q) is minimum.

In the case of ***XB-tree***, we use the function *getNext*() given in [4] to do the task and fit it for our strategy, in which the following functions are used.

isLeaf(q) - returns true if q is a leaf of Q; otherwise, false.

isRoot(q) - returns true if q is the root of Q; otherwise, false.

currL(σ_q) - return the *LeftPos* of the entry pointed to by σ_q.

currR(σ_q) - returns the *RightPos* of the entry pointed to by σ_q.

isPlainValue(σ_q) - returns true if σ_q is pointing to a leaf node in the corresponding ***XB-tree***.

end(Q) - if for each leaf node q of Q, $\sigma_q = \phi$ (i.e., $B(q)$ is exhausted), then returns true; otherwise, false.

Fuction *getNext*(q) (*Initially, q is the root of Q.*)

begin

1: **if** (*isLeaf*(q)) **then** return q;
2: **for** each child q_i of q **do**
3: { $r_i \leftarrow getNext(q_i)$;
4: **if** ($r_i \neq q_i \vee \neg isPlainValue(\sigma_q)$) **then** return q;
5: $q_{min} \leftarrow q''$ such that $currL(\sigma_{q''}) = min_i\{currL(\sigma_{r_j})\}$;
6: $q_{max} \leftarrow q'''$ such that $currL(\sigma_{q'''}) = max_i\{currL(\sigma_{r_i})\}$;
7: **while** ($currR(\sigma_q) < currL(\sigma_{q_{max}})$) do *advance* ($\sigma_q$);
8: **if** ($currL(\sigma_q) < currL(\sigma_{q_{min}})$) **then** return q;
9: **else** return q_{min};
10: }

end

The goal of the above function is to figure out a query node to determine what entry from data streams will be checked in the next step, which has to satisfy the above condition (i) - (iii). Lines 7 - 9 are used to find a query node satisfying condition (i) (see Figure 3.10 for illustration of Line 7). The recursive call performed in line 3 shows that condition (ii) is met. Since each ***XB-tree*** is navigated top-down and the entries in each node are scanned form left to right, condition (iii) must be always satisfied.

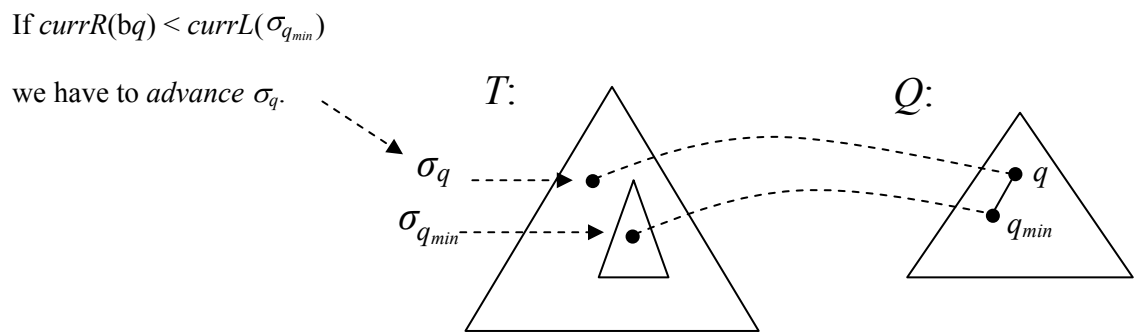


Figure 3.10 Illustration for advance (σ_q)

Once a $q \in Q$ is returned, we will further check σ_q. If it is an entry in a leaf node in the corresponding ***XB-tree***, insert it into stack *ST* (See Algorithm *stream-transformation*().) Otherwise, we will do *advance* (σ_q) or *drilldown*(σ_q), according to the relation ship between σ_q and the nodes stored in *ST*.

We associate each $q \in Q$ with an extra linked list, denoted $link_q$, such that each entry in it contains a pointer to a node v stored in *ST* with label(v) = label(q). We append entries to the end of a $link_q$ one by one as the document nodes are inserted into *ST*, as illustrated in Figure 3.11(a). The last entry in $link_q$ is denoted as $link_{qlast}$.

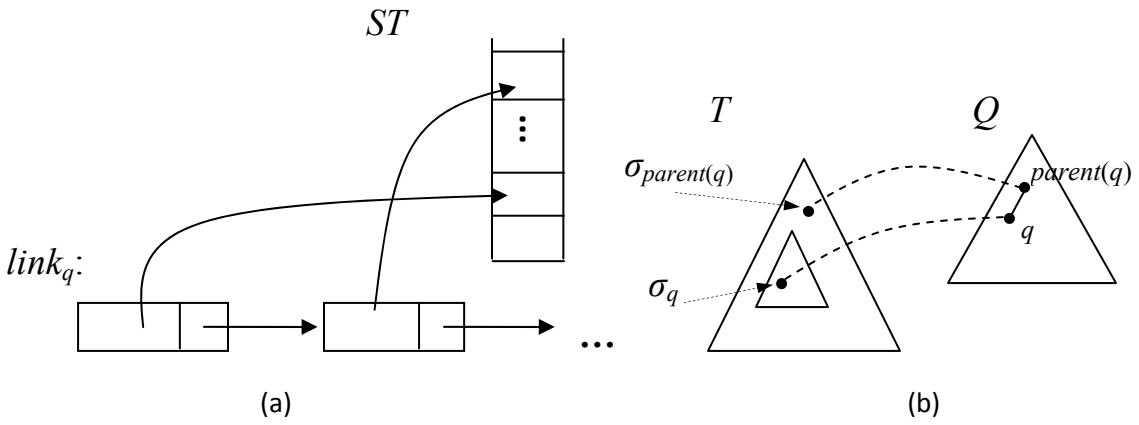


Figure 3.11 Illustration for advance (σ_q)

Based on the simple structure, *tree-embedding*() is modified as follows.

Algorithm *tree-embeddingXB*(*Q*)

---

begin

1: **while** ($\neg$*end*(*Q*)) **do**
2: {$q \leftarrow$ *getNext*(*root-of-Q*);
3: **if** (*isPlainValue*(σ_q) **then**;
4: {let *v* be the node pointed to by σ_q;
5: **while** *ST* is not empty and *ST.top* is not *v*'s ancestor **do**
6: {$x \leftarrow$ *ST.pop*(); Let $x = (q', u)$; (*a node for *u* will be created.*)
7: call *embeddingCheck*(*q'*, *u*); }
8: *ST.push*(*q*, *v*); *advance*(σ_q);
9: }
10: **else if** (($\neg$*isRoot*(*q*) $\wedge$ *linkq* $\neq \phi \wedge$ *currR*(σ_q) < LeftPos($link_{q,last}$)
11: **then** *advance*(σ_q) (*not part of a solution*)
12: **else** *drilldown*(σ_q); (*may find a solution.*)
}
end

In the above algorithm, we distinguish between two cases. If σ_q) is a leaf node in the corresponding ***XB-tree***, we will insert it into ***ST***. Otherwise, lines 10 - 12 will be carried out. If currR(σ_q) < LeftPos($link_{parent(q),last}$), we have a situation as illustrated in Figure 3.11(b). In this case, we will advance σ_q (see line 11.) If it is not the case, we will drill down the corresponding ***XB-tree*** (see line 2) since a solution may be found.

Chapter 4

Performance Evaluation

In this section, we will present the results of the experimental evaluation of the proposed tree pattern matching algorithms. In particular, we evaluate the performance on several data sets and compare our algorithms with some other twig matching algorithms reviewed in chapter 2. Then, we focus on our algorithms again and discuss the main advantage of our algorithms.

4.1 Experimental Setup

We implemented our ***TreeEmbed*** algorithm using C++ and performed on a Pentium IV 3.0Ghz PC with 2GB RAM and 80 GB hard disk, running Windows XP professional with Service Pack 3. We compare ***TreeEmbed*** with 3 other twig join algorithms: ***TwigStack*** [4], ***Twig^{2}Stack*** [7], and ***TwigList*** [10]. We choose ***TwigStack*** as the basis for comparison, as it is the classical holistic twig join algorithm. ***Twig^{2}Stack*** improves ***TwigStack*** by using a kind of complex data structure for storing intermediate results. ***TwigList*** simplifies the data structure used by ***Twig^{2}Stack***, which is useful from a practical viewpoint.

4.2 Data Sets

Our experiments are based on both real and synthetic data sets. For real data we use TreeBank from [11] and DBLP [13]. For synthetic data we use Xmark from [12].

- The TreeBank [11] is a project which focus on the text corpora structure analyze. In linguistics, in order to statistically analyze language structure, we need to annotate a corpus by *POS-tagging* (Part of speech tagging). For example, information about each word's part of speech is tagged by *verb*, *noun*, *adjective*, etc. Commonly, the structures are represented as tree structures, as shown in Figure 4.1. The deep recursive structure of this data makes it an interesting case for experiments.

- The Digital Bibliography and Library Project database (DBLP) is the popular computer science bibliography in the XML format. It includes conference paper articles, journal papers, etc. The original data set is a huge file with file size 650MB. DBLP dataset is a wide and shallow document, as shown in Figure 4.1.

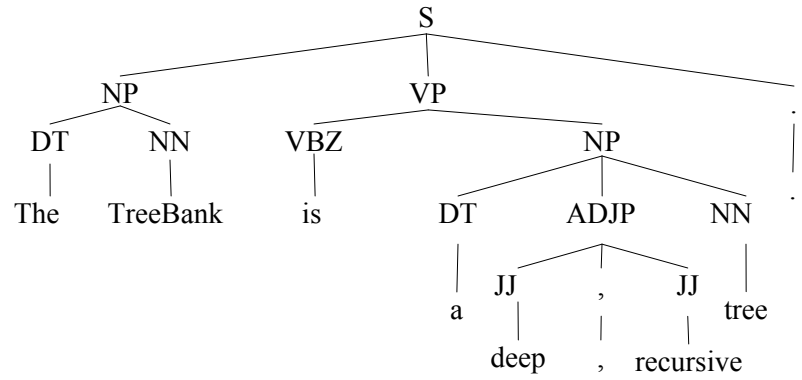
- XMark is an XML benchmark project. It can efficiently generate XML document files with several different scales, which can be the size of several GBs. Independent of the size of generated documents; it uses only low and constant memory. We use this data generator to generate several synthetic dataset for scalability analysis.

Some quantitative characteristics of the data sets are summarized in Table 1.

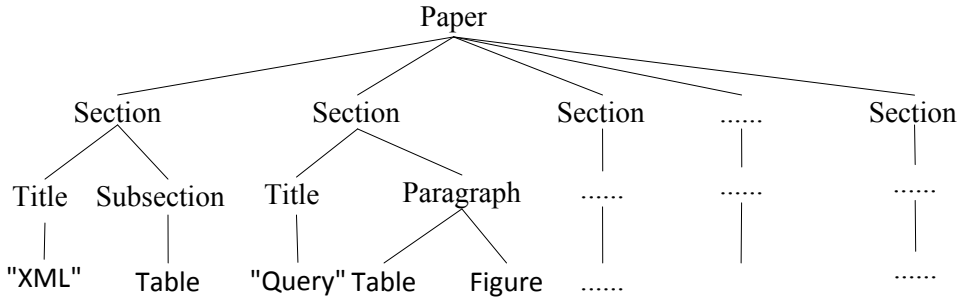
To study the effects of different tree shapes, we use TreeBank data set. To explore the impact of document quantity, we use DBLP. Xmark is for checking scalability.

Table 4.1 The List of Data Sets

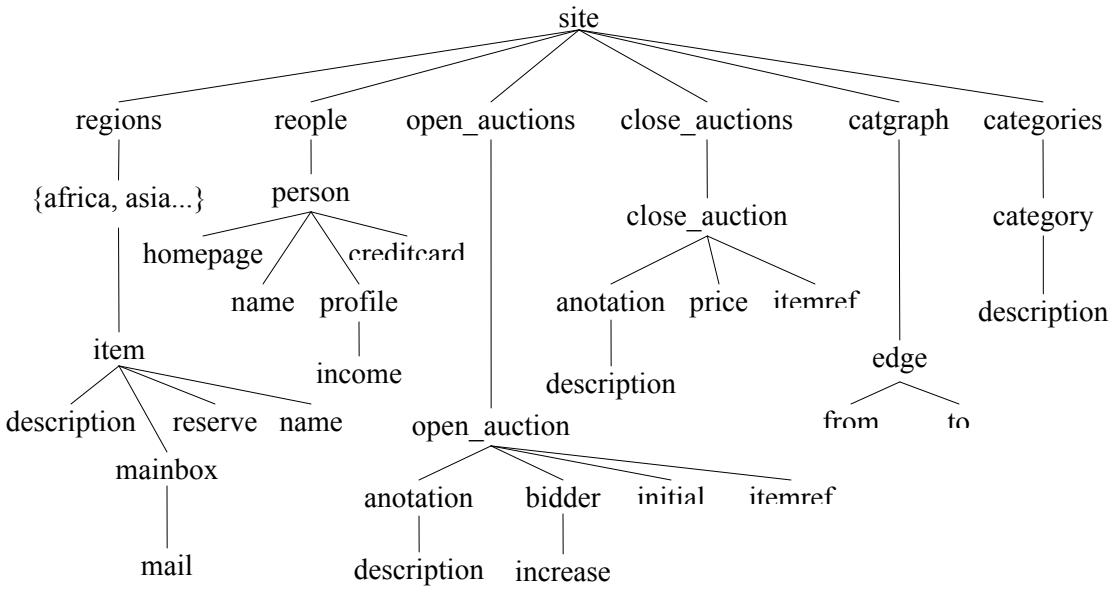
	Size of Data Set(MB)	**Number of Nodes(Million)**	**Max Tree Depth**	**Average Tree Depth**
TreeBank	82	2.4	36	7.9
DBLP	650	16.8	7	2.7
XMark1	151	2	12	5.5
XMark2	303	4.1	11	5.0
XMark3	456	6.1	12	5.5
XMark4	609	8.2	11	5.5
XMark5	761	10.2	11	5.0



(a) A sample of **TreeBank** data structure



(b) A sample of **DBLP** data structure



(c) A sample of **XMark** data structure

Figure 4.1 Sample data structures of 3 different data sets

4.3 Tested Methods

We experimented with the following four methods of twig pattern query evaluation to study and compare their performance.

TwigStack [4]: The first holistic twig join algorithm. It uses a chain of linked stacks to compactly represent partial results, which match the root-to-leaf query paths. They are then composed to obtain matches for the twig pattern.

Twig^{2}Stack [7]: A bottom up algorithm for processing twig queries based on an encoding scheme. The algorithm generates a single combined stream with post order sorting for all query nodes matches by using a single stack. Working in the postorder, it can be decided if an entire subtree has a match when the top node is met.

TwigList [10]: One-phase holistic twig pattern matching algorithms based on *TwigStack*. It avoids devising a stack structure to hold matching paths until all twig matches are formed, by using a simple list and intervals given by pointers.

TreeEmbed [5]: Processing a tree reconstruction from data streams with the XB-tree index structure being used. It associates each query node with an XML data stream during the reconstruction process to reduce the time complexity.

4.4 Experiments on TreeBank

In this section, we present our test results on the TreeBank data set. It is a simple data set of sentence structures with each represented as a tree. In each tree, the leaf nodes are the words in a sentence, and the root node as well as the internal nodes, represent the structure of the sentence, as shown in Figure 4.1(a). A sentence can be very complicated and the tree representing it can be deep and recursive, which makes the data set an interesting case for experiments. In the test, we use a variety of XML queries patterns, as shown in Table 4.2- 4.6.

4.4.1 Queries

We tested 25 queries which are organized into 5 groups as shown in Table 4.2 - 4.6. The syntax of the path expressions is borrowed form XPath, and is simplified for the sake of easy understanding. In an expression, '/' stands for a parent-child relationship, and '//' for an ancestor-descendant relationship. The expression within a pair of square brackets is a predicate. The logic symbol "∧" connects different paths together.

Table 4.2 Group I. Queries with incremental path lengths.

Query	Path Expression
Q1	//S//NP
Q2	//S//NP//NNP
Q3	//S//NP//VP//NNP
Q4	//S//VP//ADJP//S//NNP
Q5	//S//VP//ADJP//SBAR//S//NNP

Table 4.3 Group II. Queries with incremental depths.

Query	Path Expression
Q6	//S//NP
Q7	//S[.//NNP ∧ VP]
Q8	//S[.//NNP]/VP[.//NP]
Q9	//S[.//NNP]/VP[.//NP[.//S]]
Q10	//S[.//NNP]/VP[.//NP[.//S[.//NNP]]]

Table 4.4 Group III. Queries matching at higher level of a document.

Query	Path Expression
Q10	//S [NP/NNP]/VP[VBD]
Q11	//S [NP/NNP]/VP[VBZ]
Q12	//S [NP/NN]/VP[VBD]
Q13	//S [NP/NN]/VP[VBZ]
Q14	//S [NP/PRP]/VP[VBD]

Table 4.5 Group IV. Queries matching at middle level of a document.

Query	Path Expression
Q16	//NP [NP/NNP]/VP[VBD]
Q17	//NP [NP/NNP]/VP[VBZ]
Q18	//NP [NP/NN]/VP[VBD]
Q19	//NP [NP/NN]/VP[VBZ]
Q20	//NP [NP/PRP]/VP[VBD]

Table 4.6 Group V. Queries matching at lower levels of a document.

Query	Path Expression
Q21	//VP [VBZ['be']]/ADVP[RB['here']]
Q22	//VP [VBZ['is']]/ADVP[RB['here']]
Q23	//VP [TO['to']]/ VP[VB['leave']]
Q24	//VP [TO['to']]/ VP[VB['rain']]
Q25	//VP [MD['should']]/ VP[VB['leave']]

The queries in Group I are used to test the impact of path lengths on performance. The queries in Group II are to test the impact of node degrees on performance. The queries in Group III - V are to test the impact on performance when query trees are embedded in different parts of a document. In a same group, the queries are embedded at the same subtree level and follow the left-to-right order.

4.4.2 Test results

We ran each group five times, and recorded an average execution time for each query as the final test result.

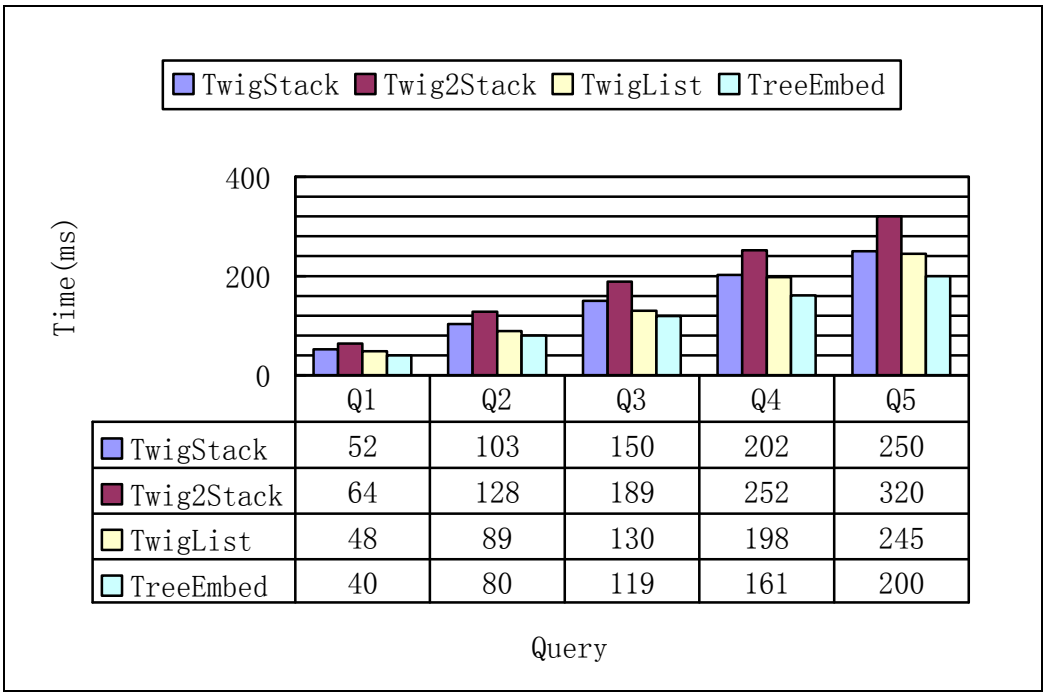
Figure 4.2 shows the test results of Group I. From the figure, we can see that. *Twig*^{2}*Stack* is less efficient than the other three algorithms. The reason for this is that, with only one single path involved, the hierarchical stack operation in *Twig*^{2}*Stack* spends some unnecessary operation time. Comparing the results shown Figure 4.2 (a), (b) and (c), we also see that ***TreeEmbed*** works better than the other 3 algorithms, especially the total execution time is much lower than theirs. The Figure 4.2(c) shows the comparing checking times for each algorithm, it explains the where the time spent on when querying take place. Actually, the total execution time grows

in order of the times of comparing nodes. In our algorithm, only a tree reconstruction process is involved, in which the tree matching is checked with no join operations being performed.

The results of Group II are shown in Figure 4.3. As the query tree depth increases, the total execution time of ***TwigStack*** increases dramatically. It is because for the deep and recursive data structure of the TreeBank *getNext*() function (used in ***TwigStack***) has to a lot of checkings. We also find that our algorithm works best for this group of queries. The reason for this is that we treat the data as a stream. No matter how deep the recursion of the data structure is, we only visit each node once. By using the *XB-tree* index structure, we can avoid any unnecessary node access.

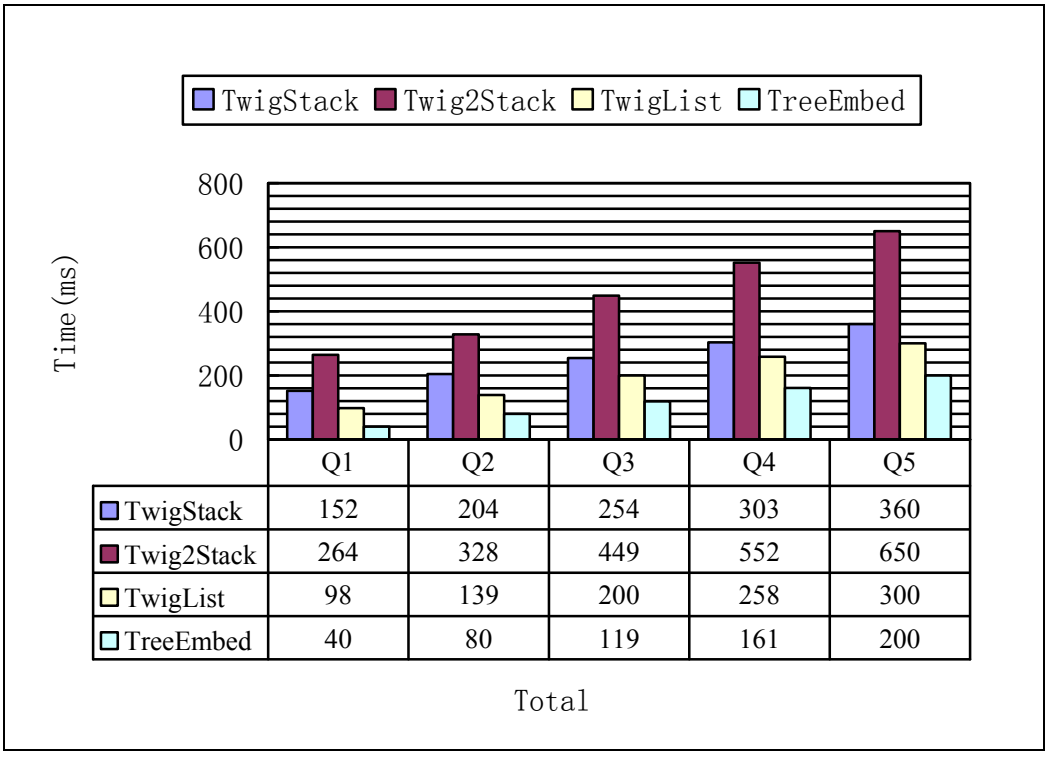
The results form the rest three groups (Figure 4.4, 4.5, 4.6) are similar to each other no matter where a matching takes place. For the same reason as above, ***TwigStack*** needs more time than the other three methods. It spends around 4 to 7 seconds for the queries in Group III, VI, and V. Again, TreeEmd method uses the least time with around 0.5 second for answering each query.

Due to the two phase operations of ***TwigStack***, it needs to store all path matches in memory. From the Figures about memory usage, we can found ***TwigStack*** consumes much more memory than the other 3 one phase algorithms.



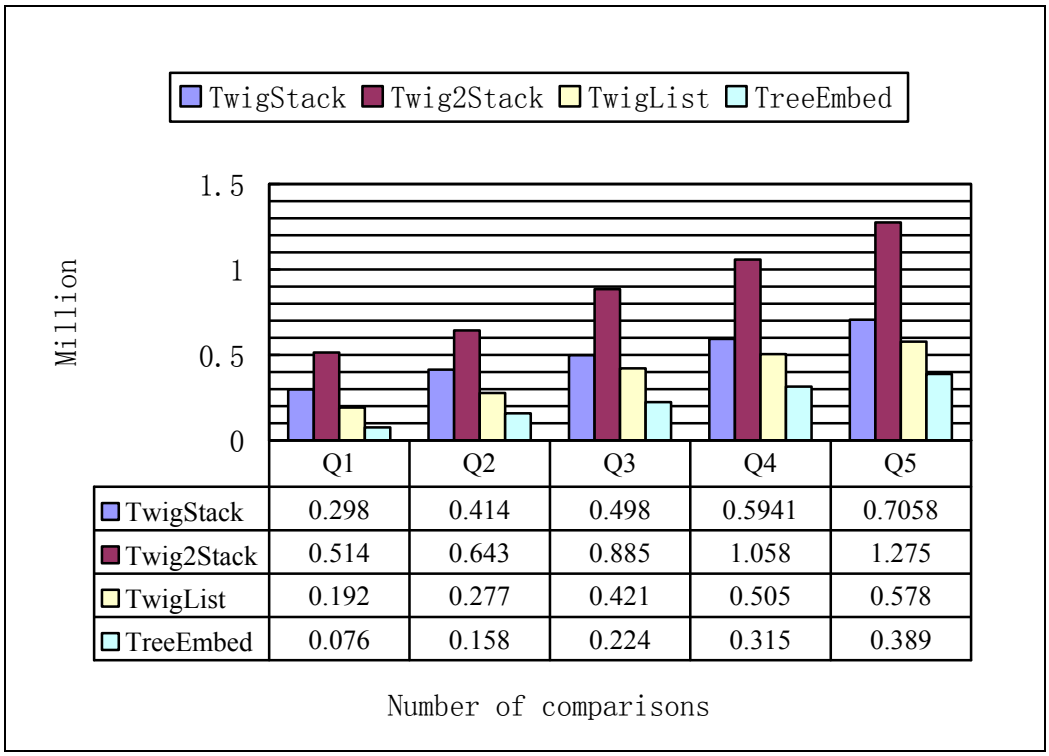
	Q1	Q2	Q3	Q4	Q5
TwigStack	52	103	150	202	250
Twig2Stack	64	128	189	252	320
TwigList	48	89	130	198	245
TreeEmbed	40	80	119	161	200

Figure 4.2 (a) Query Time in Group One



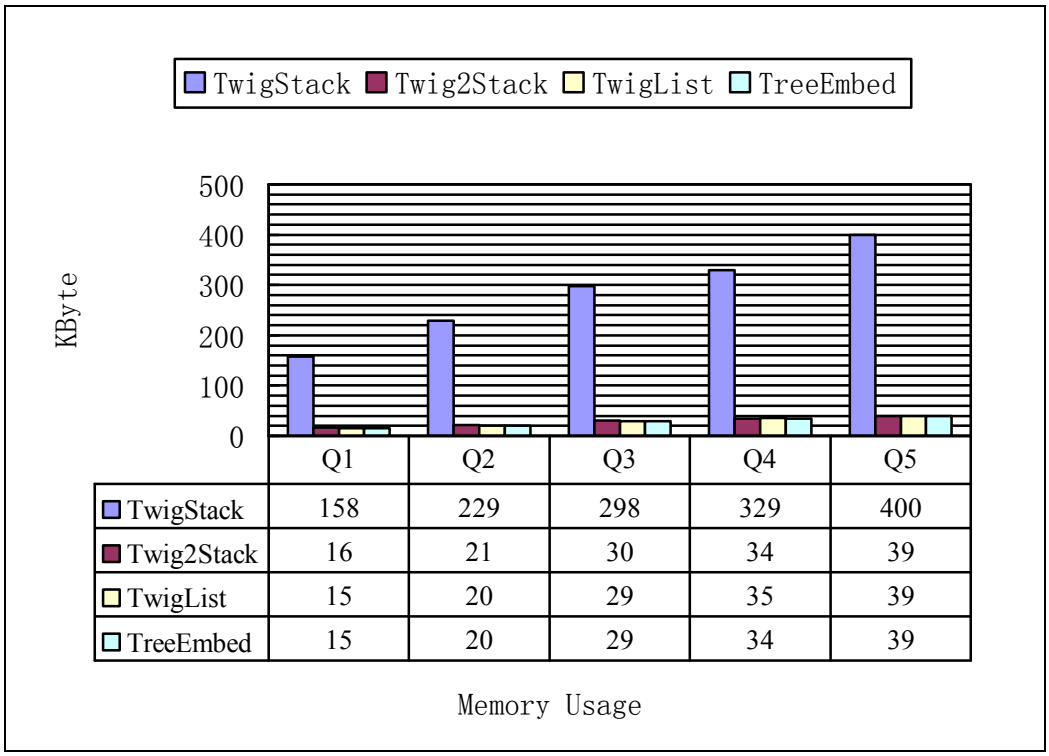
	Q1	Q2	Q3	Q4	Q5
TwigStack	152	204	254	303	360
Twig2Stack	264	328	449	552	650
TwigList	98	139	200	258	300
TreeEmbed	40	80	119	161	200

Figure 4.2 (b) Total Execution Time of Group One



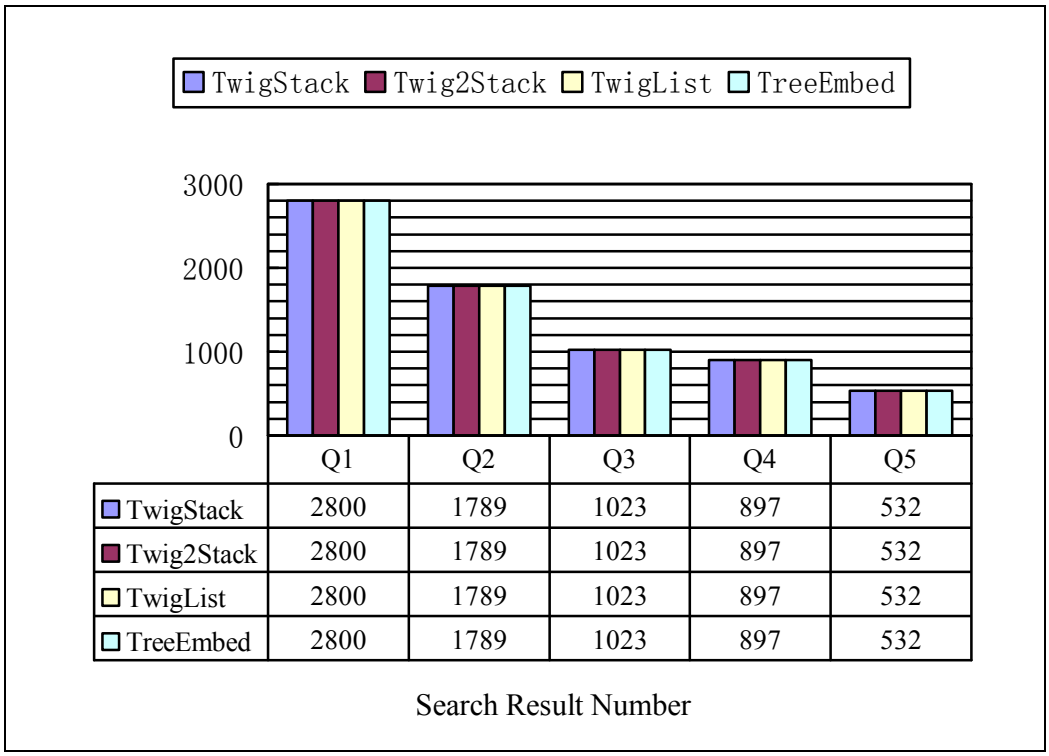
	Q1	Q2	Q3	Q4	Q5
TwigStack	0.298	0.414	0.498	0.5941	0.7058
Twig2Stack	0.514	0.643	0.885	1.058	1.275
TwigList	0.192	0.277	0.421	0.505	0.578
TreeEmbed	0.076	0.158	0.224	0.315	0.389

Figure 4.2 (c) Total number of comparisons in Group One



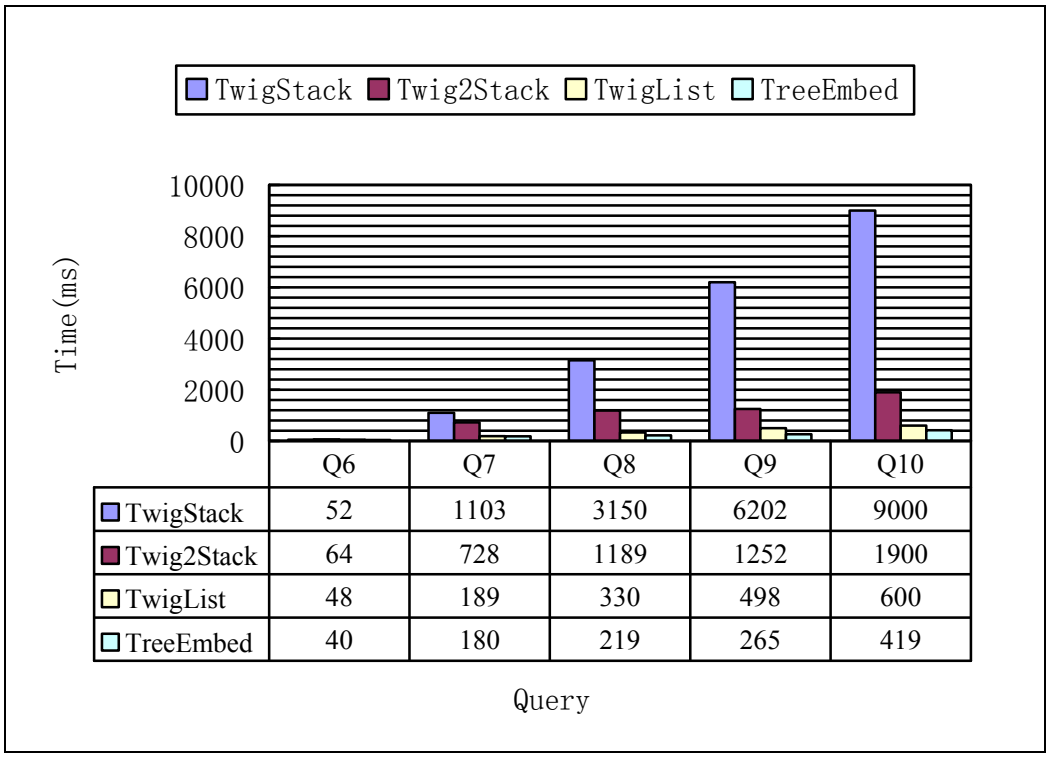
	Q1	Q2	Q3	Q4	Q5
TwigStack	158	229	298	329	400
Twig2Stack	16	21	30	34	39
TwigList	15	20	29	35	39
TreeEmbed	15	20	29	34	39

Figure 4.2 (d) Memory Usage in Group One



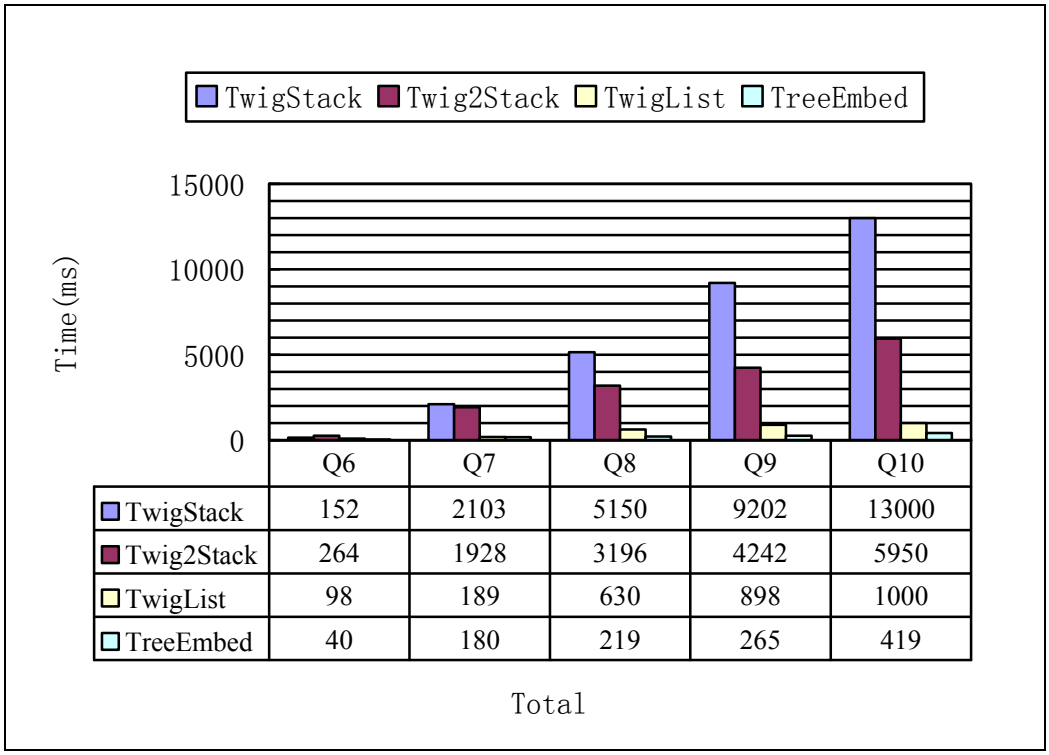
	Q1	Q2	Q3	Q4	Q5
TwigStack	2800	1789	1023	897	532
Twig2Stack	2800	1789	1023	897	532
TwigList	2800	1789	1023	897	532
TreeEmbed	2800	1789	1023	897	532

Figure 4.2 (e) Number of Search Results in Group One



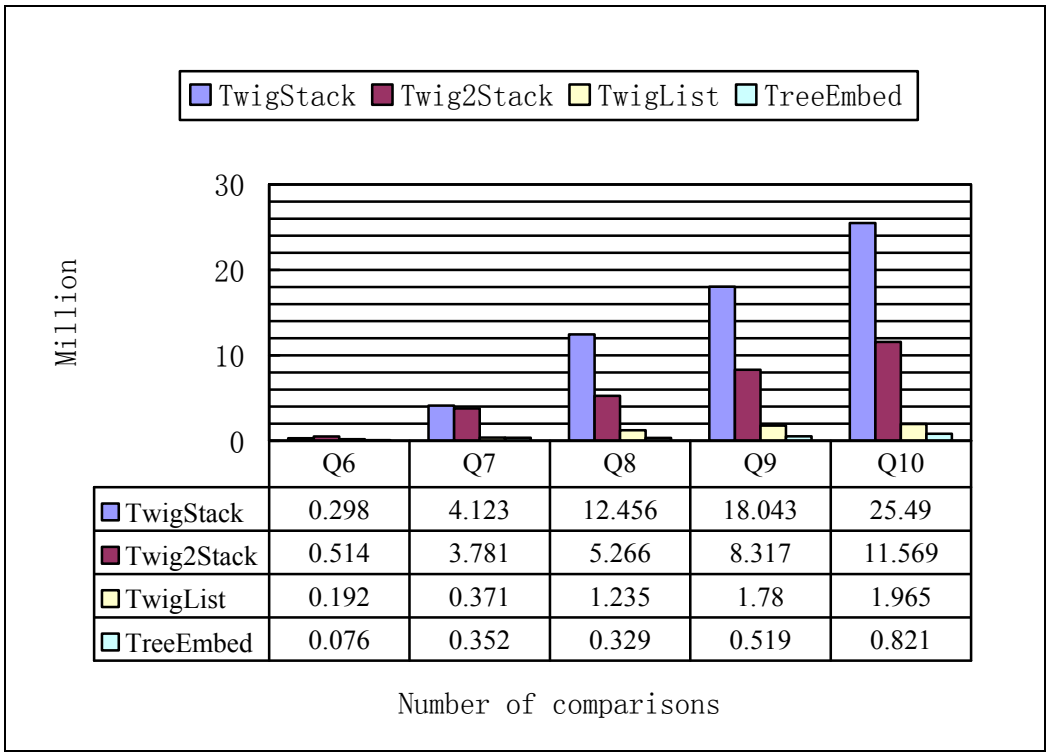
	Q6	Q7	Q8	Q9	Q10
TwigStack	52	1103	3150	6202	9000
Twig2Stack	64	728	1189	1252	1900
TwigList	48	189	330	498	600
TreeEmbed	40	180	219	265	419

Figure 4.3 (a) Query time in Group Two



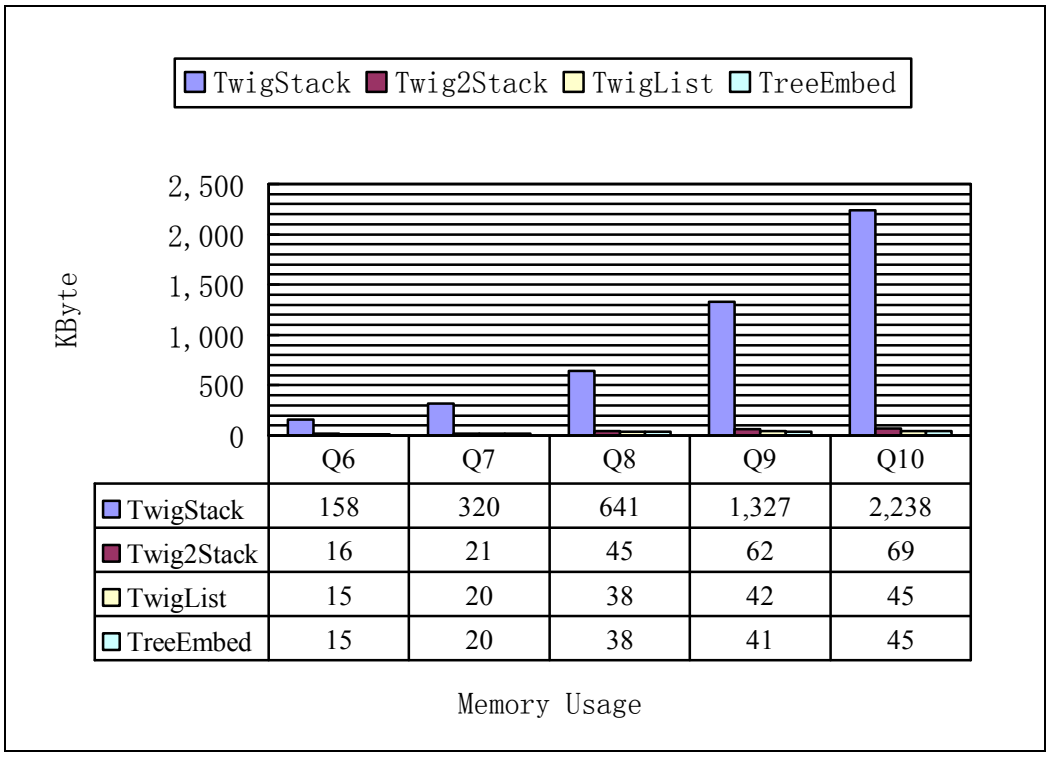
	Q6	Q7	Q8	Q9	Q10
TwigStack	152	2103	5150	9202	13000
Twig2Stack	264	1928	3196	4242	5950
TwigList	98	189	630	898	1000
TreeEmbed	40	180	219	265	419

Figure 4.3 (b) Total Execution Time of Group Two



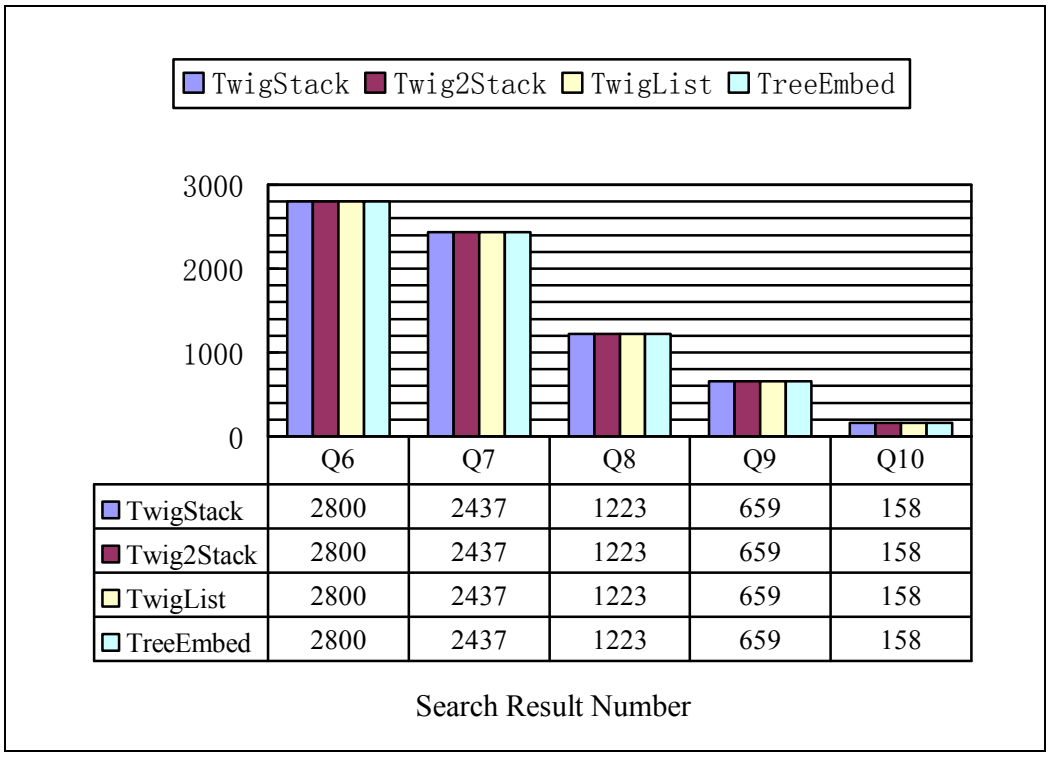
	Q6	Q7	Q8	Q9	Q10
TwigStack	0.298	4.123	12.456	18.043	25.49
Twig2Stack	0.514	3.781	5.266	8.317	11.569
TwigList	0.192	0.371	1.235	1.78	1.965
TreeEmbed	0.076	0.352	0.329	0.519	0.821

Figure 4.3 (c) Total number of comparisons in Group Two



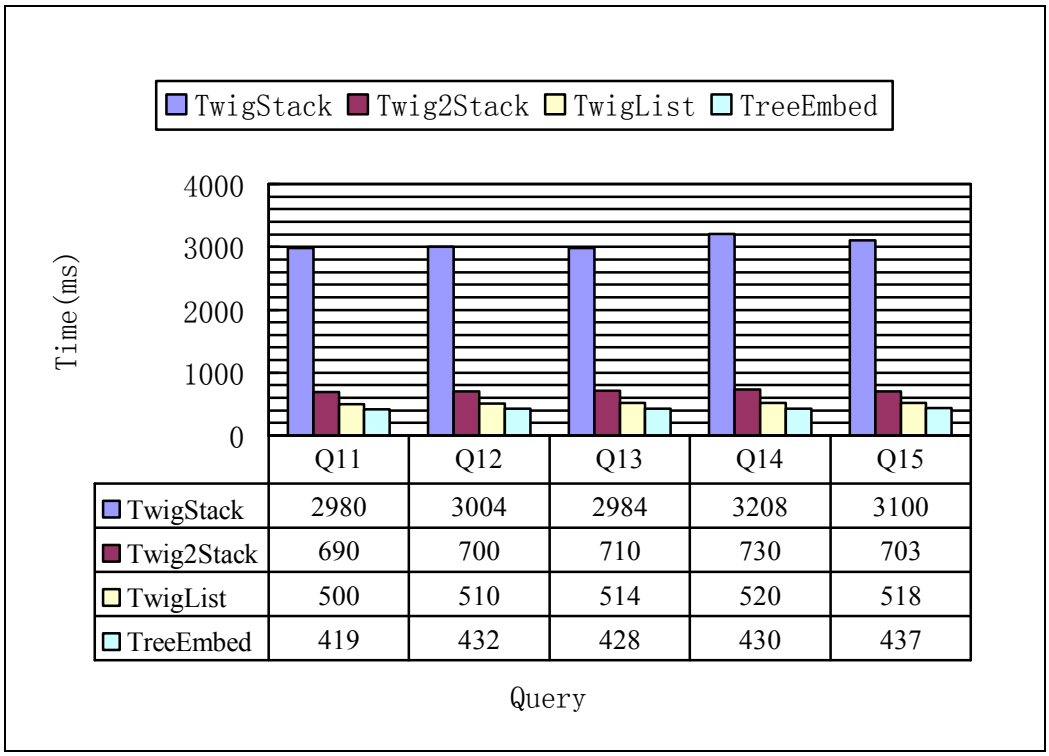
	Q6	Q7	Q8	Q9	Q10
TwigStack	158	320	641	1,327	2,238
Twig2Stack	16	21	45	62	69
TwigList	15	20	38	42	45
TreeEmbed	15	20	38	41	45

Figure 4.3 (d) Memory Usage in Group Two



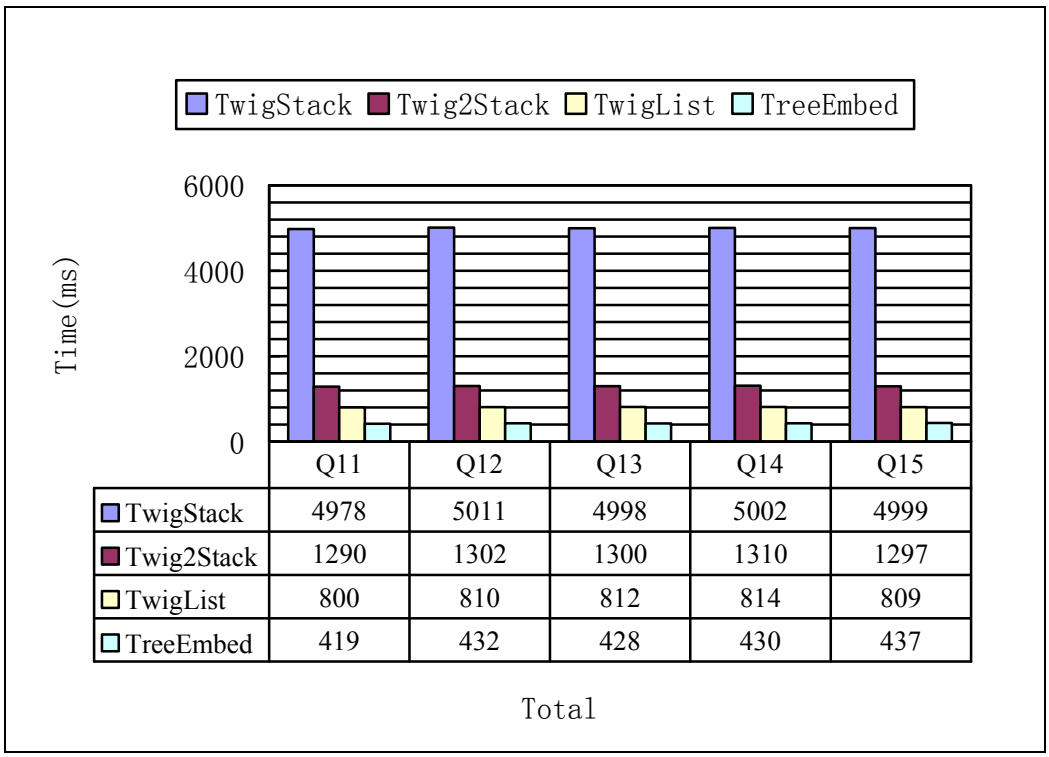
	Q6	Q7	Q8	Q9	Q10
TwigStack	2800	2437	1223	659	158
Twig2Stack	2800	2437	1223	659	158
TwigList	2800	2437	1223	659	158
TreeEmbed	2800	2437	1223	659	158

Figure 4.3 (e) Number of Search Results in Group Two



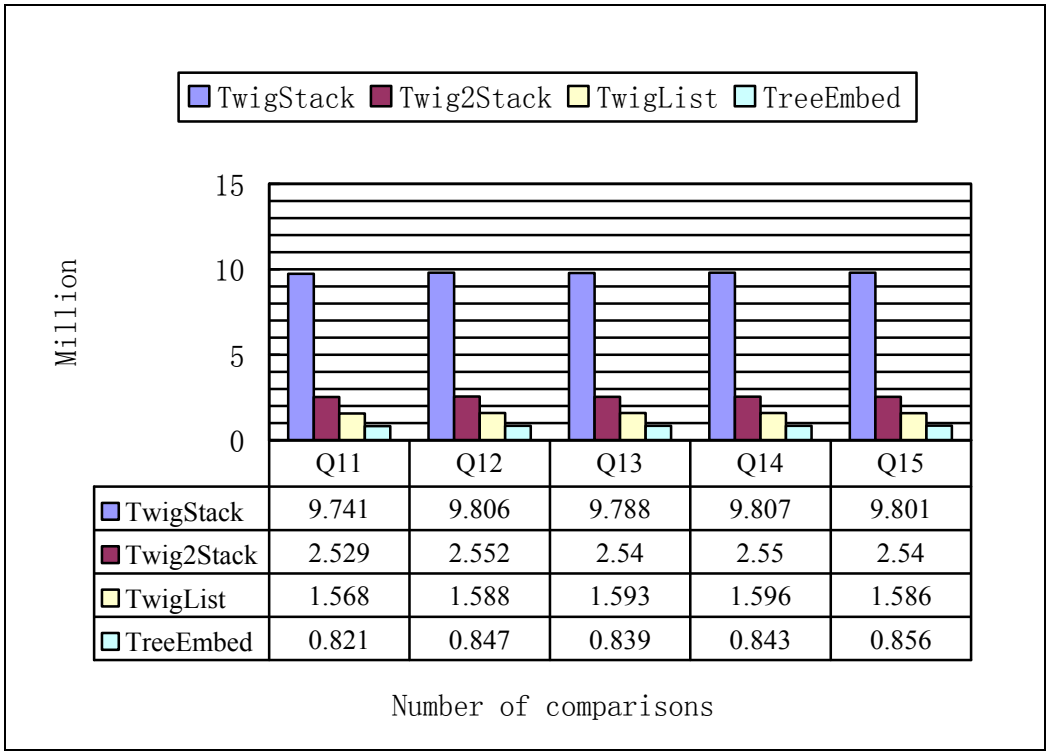
	Q11	Q12	Q13	Q14	Q15
TwigStack	2980	3004	2984	3208	3100
Twig2Stack	690	700	710	730	703
TwigList	500	510	514	520	518
TreeEmbed	419	432	428	430	437

Figure 4.4 (a) Query Time in Group Three



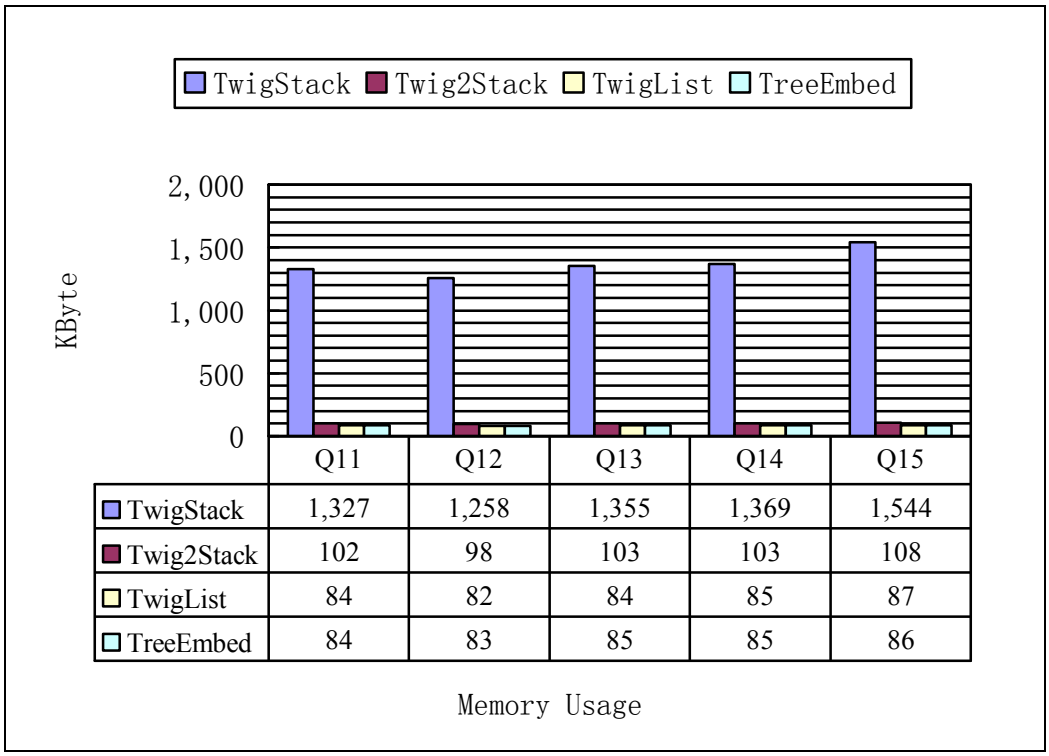
	Q11	Q12	Q13	Q14	Q15
TwigStack	4978	5011	4998	5002	4999
Twig2Stack	1290	1302	1300	1310	1297
TwigList	800	810	812	814	809
TreeEmbed	419	432	428	430	437

Figure 4.4 (b) Total Execution time in Group Three



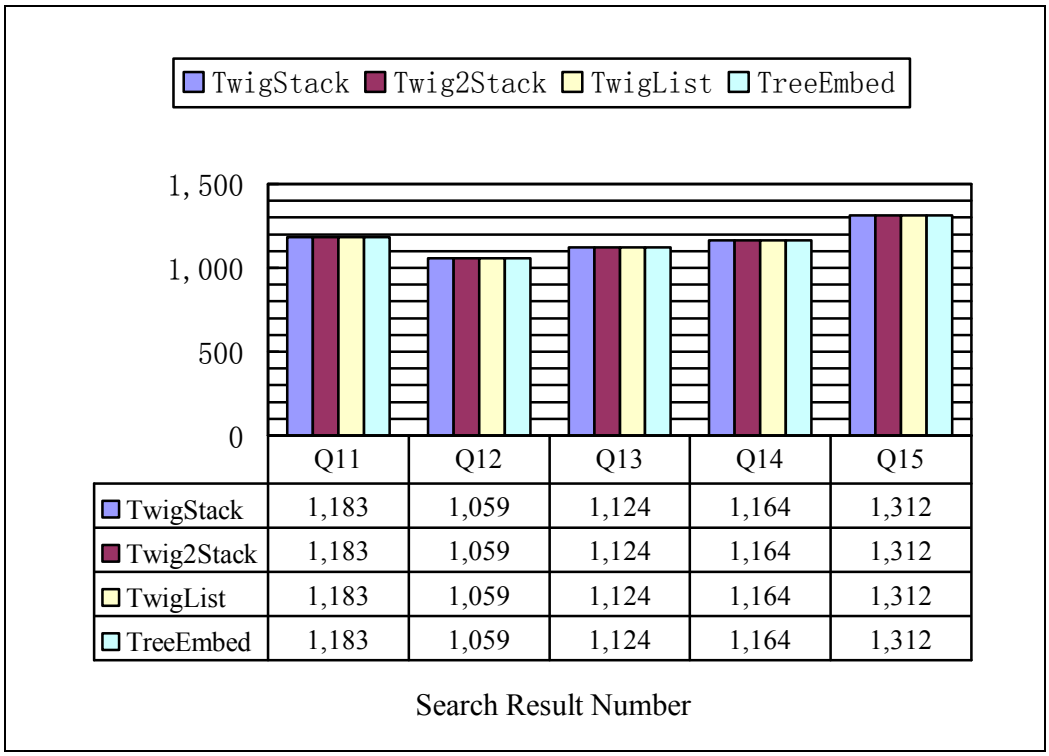
	Q11	Q12	Q13	Q14	Q15
TwigStack	9.741	9.806	9.788	9.807	9.801
Twig2Stack	2.529	2.552	2.54	2.55	2.54
TwigList	1.568	1.588	1.593	1.596	1.586
TreeEmbed	0.821	0.847	0.839	0.843	0.856

Figure 4.4 (c) Total number of comparisons in Group Three



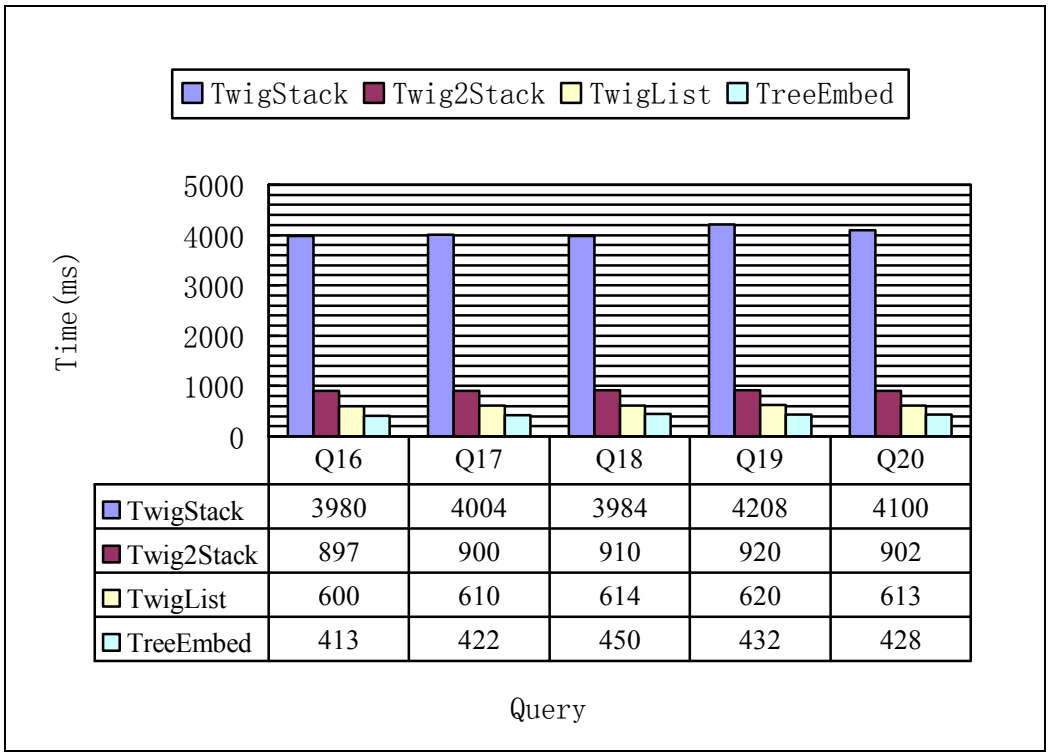
	Q11	Q12	Q13	Q14	Q15
TwigStack	1,327	1,258	1,355	1,369	1,544
Twig2Stack	102	98	103	103	108
TwigList	84	82	84	85	87
TreeEmbed	84	83	85	85	86

Figure 4.4 (d) Memory Usage in Group Three



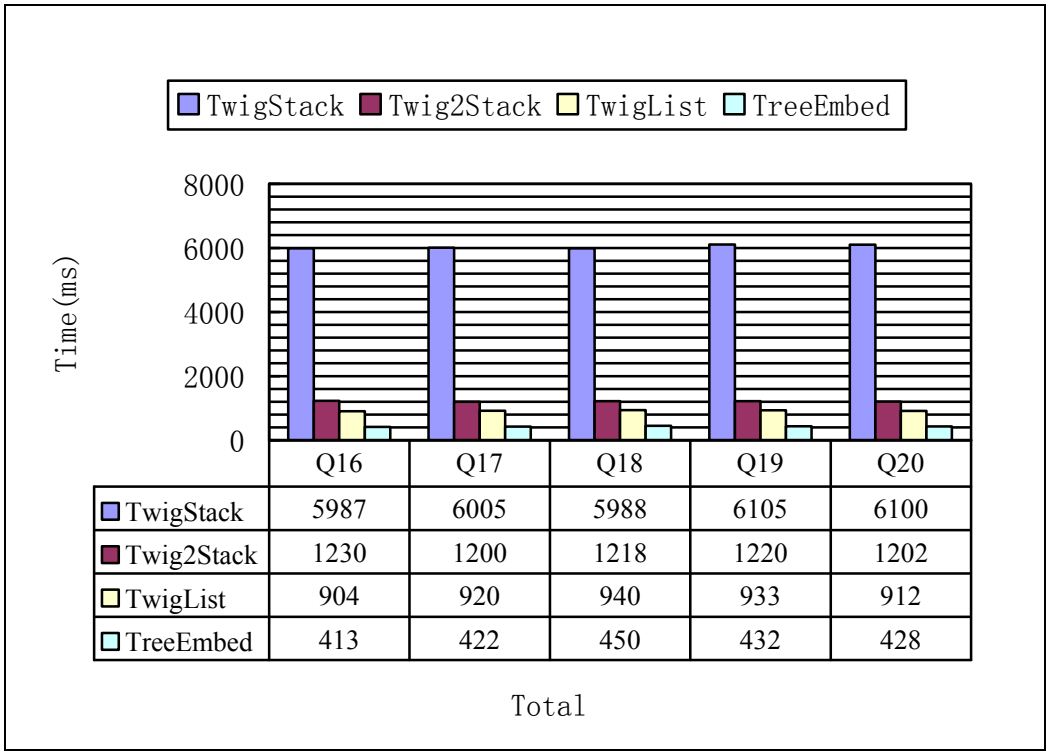
	Q11	Q12	Q13	Q14	Q15
TwigStack	1,183	1,059	1,124	1,164	1,312
Twig2Stack	1,183	1,059	1,124	1,164	1,312
TwigList	1,183	1,059	1,124	1,164	1,312
TreeEmbed	1,183	1,059	1,124	1,164	1,312

Figure 4.4 (e) Number of Search Results in Group Three



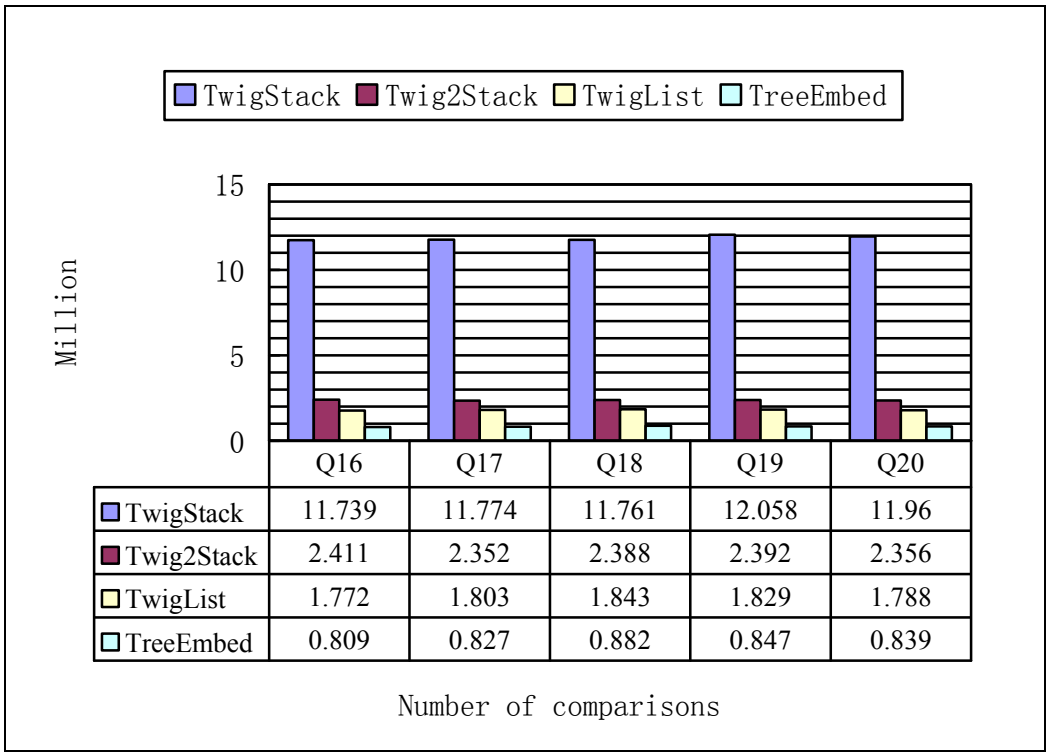
	Q16	Q17	Q18	Q19	Q20
TwigStack	3980	4004	3984	4208	4100
Twig2Stack	897	900	910	920	902
TwigList	600	610	614	620	613
TreeEmbed	413	422	450	432	428

Figure 4.5 (a) Query Time in Group Four



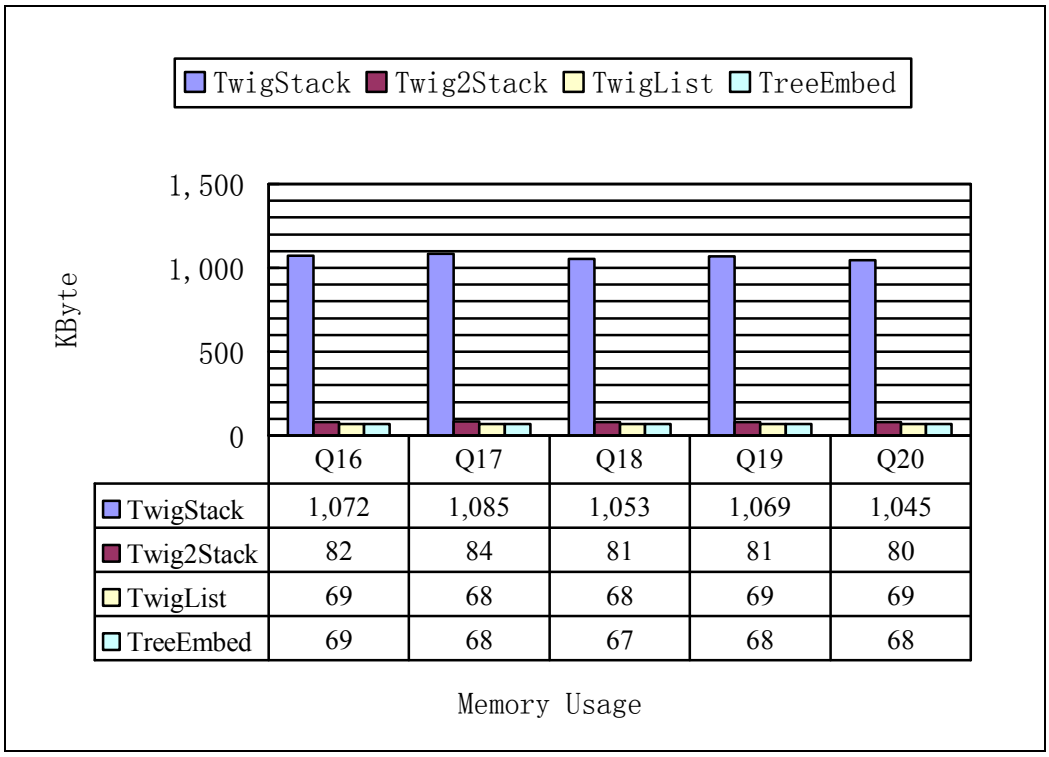
	Q16	Q17	Q18	Q19	Q20
TwigStack	5987	6005	5988	6105	6100
Twig2Stack	1230	1200	1218	1220	1202
TwigList	904	920	940	933	912
TreeEmbed	413	422	450	432	428

Figure 4.5 (b) Total Execution time in Group Four



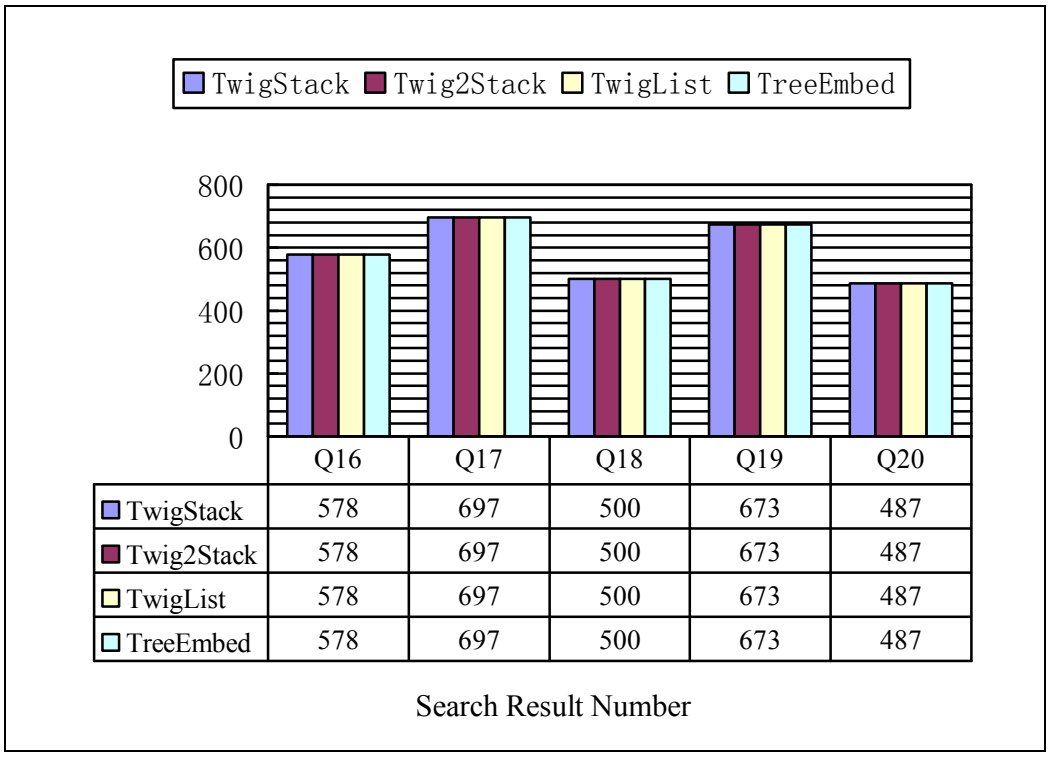
	Q16	Q17	Q18	Q19	Q20
TwigStack	11.739	11.774	11.761	12.058	11.96
Twig2Stack	2.411	2.352	2.388	2.392	2.356
TwigList	1.772	1.803	1.843	1.829	1.788
TreeEmbed	0.809	0.827	0.882	0.847	0.839

Figure 4.5 (c) Total number of comparisons in Group Four



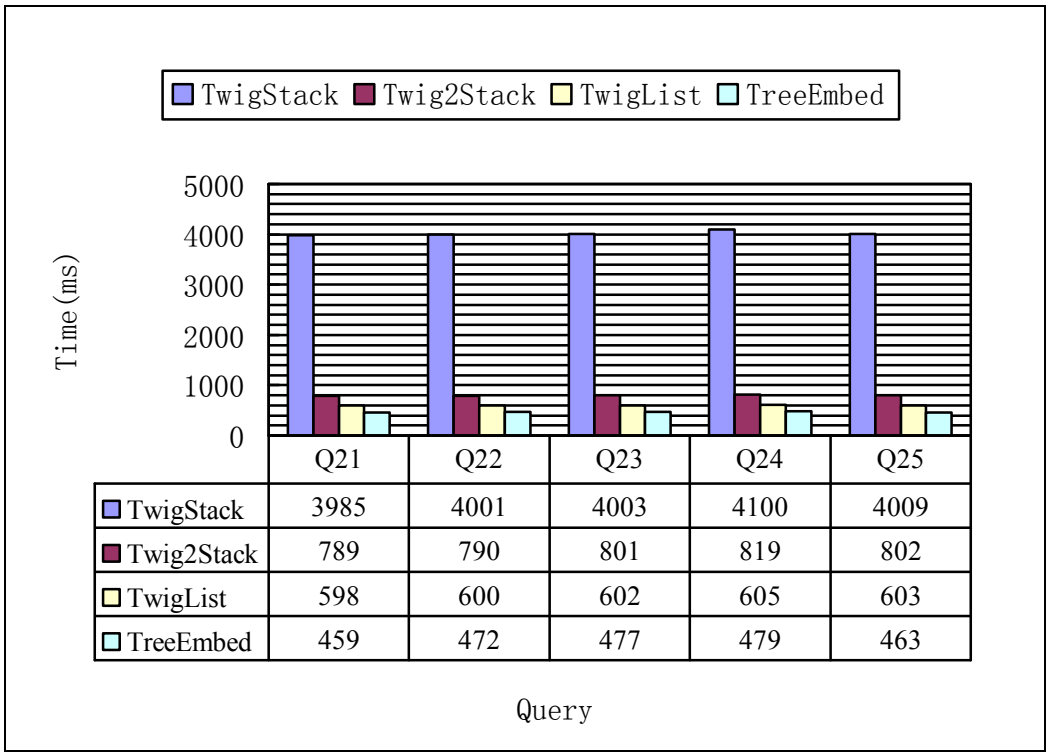
	Q16	Q17	Q18	Q19	Q20
TwigStack	1,072	1,085	1,053	1,069	1,045
Twig2Stack	82	84	81	81	80
TwigList	69	68	68	69	69
TreeEmbed	69	68	67	68	68

Figure 4.5 (d) Memory Usage in Group Four



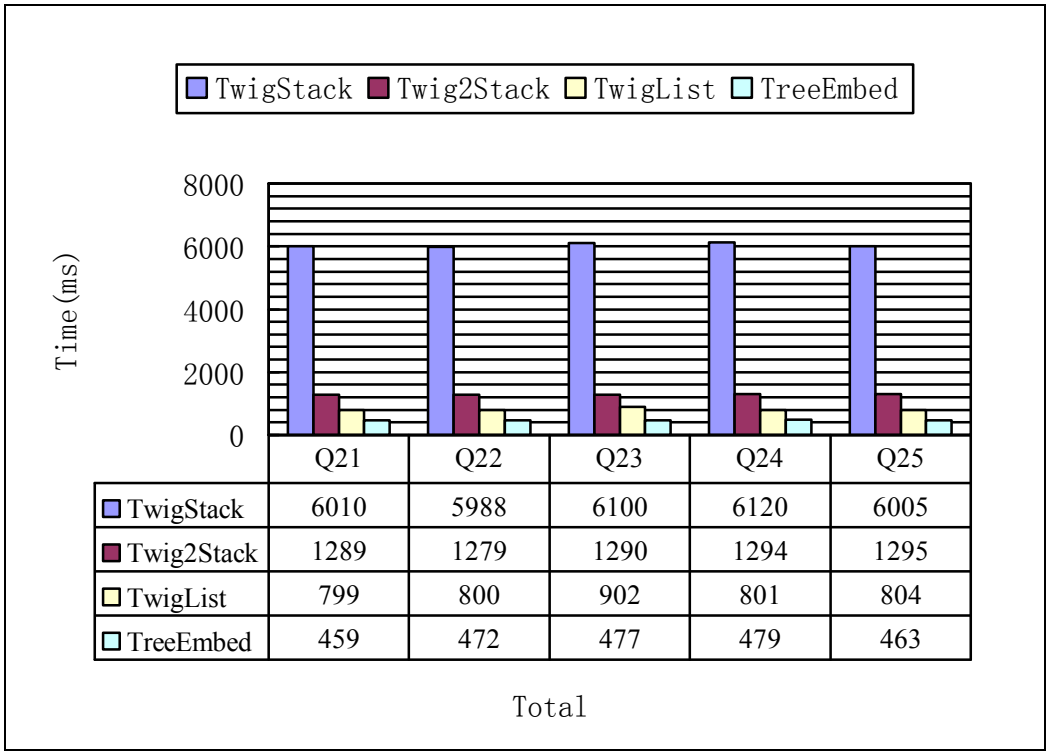
	Q16	Q17	Q18	Q19	Q20
TwigStack	578	697	500	673	487
Twig2Stack	578	697	500	673	487
TwigList	578	697	500	673	487
TreeEmbed	578	697	500	673	487

Figure 4.5 (e) Number of Search Results in Group Four



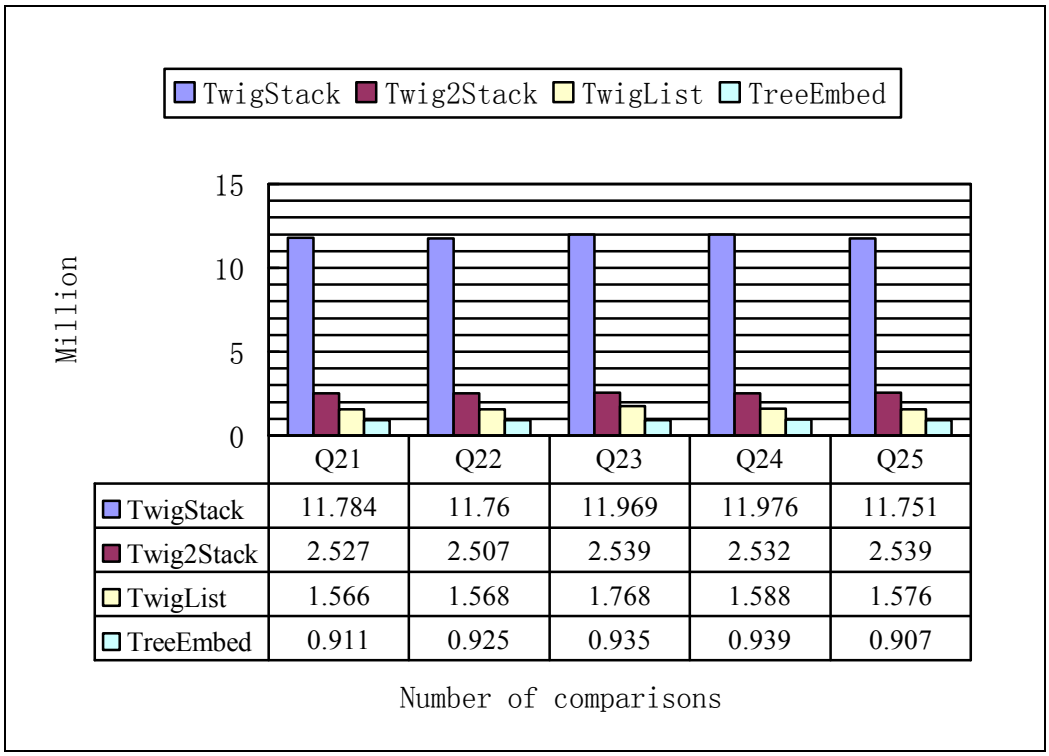
	Q21	Q22	Q23	Q24	Q25
TwigStack	3985	4001	4003	4100	4009
Twig2Stack	789	790	801	819	802
TwigList	598	600	602	605	603
TreeEmbed	459	472	477	479	463

Figure 4.6 (a) Query Time in Group Five



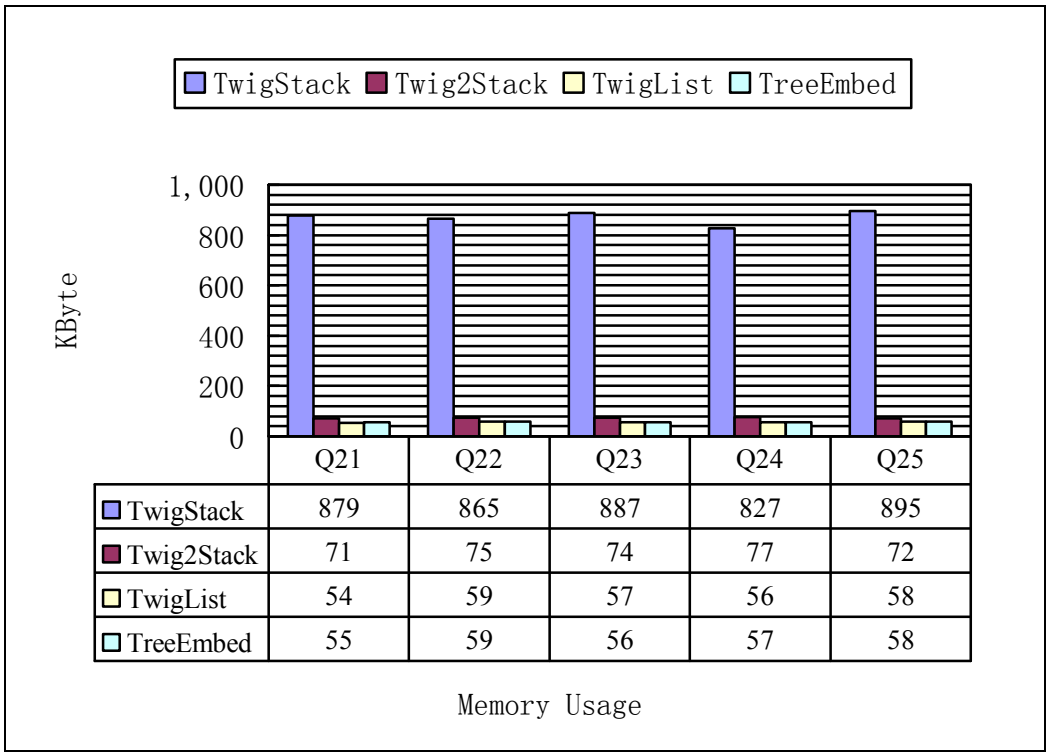
	Q21	Q22	Q23	Q24	Q25
TwigStack	6010	5988	6100	6120	6005
Twig2Stack	1289	1279	1290	1294	1295
TwigList	799	800	902	801	804
TreeEmbed	459	472	477	479	463

Figure 4.6 (b) Total Execution time in Group Five



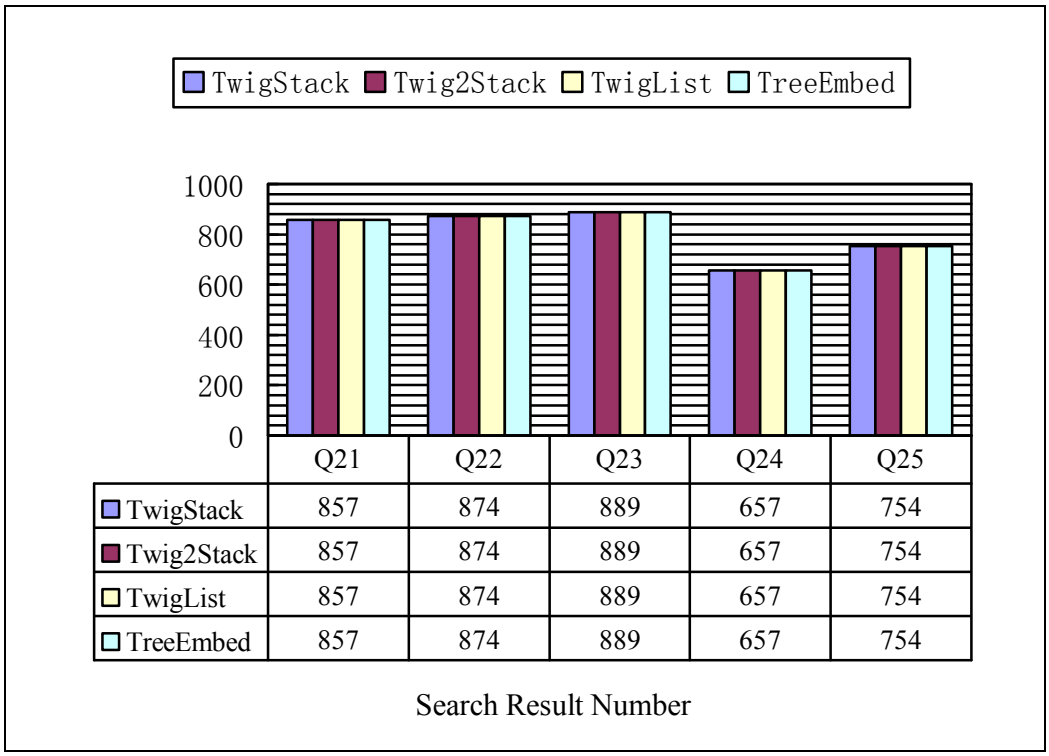
	Q21	Q22	Q23	Q24	Q25
TwigStack	11.784	11.76	11.969	11.976	11.751
Twig2Stack	2.527	2.507	2.539	2.532	2.539
TwigList	1.566	1.568	1.768	1.588	1.576
TreeEmbed	0.911	0.925	0.935	0.939	0.907

Figure 4.6 (c) Total number of comparisons in Group Five



	Q21	Q22	Q23	Q24	Q25
TwigStack	879	865	887	827	895
Twig2Stack	71	75	74	77	72
TwigList	54	59	57	56	58
TreeEmbed	55	59	56	57	58

Figure 4.6 (d) Memory Usage in Group Five



	Q21	Q22	Q23	Q24	Q25
TwigStack	857	874	889	657	754
Twig2Stack	857	874	889	657	754
TwigList	857	874	889	657	754
TreeEmbed	857	874	889	657	754

Figure 4.6 (e) Number of Search Results in Group Five

4.5 Experiment on DBLP data set

In this experiment, we report the test results on DBLP data set. DBLP is a wide and shallow data set. It is very suitable for us to test the quality of the four different kind of twig pattern matching algorithm.

4.5.1 Queries

In this section, we list all the queries used in the test. They can be organized into three groups of queries: small, median and large. Each group has 5 different queries, in which the query node names are different, as shown below:

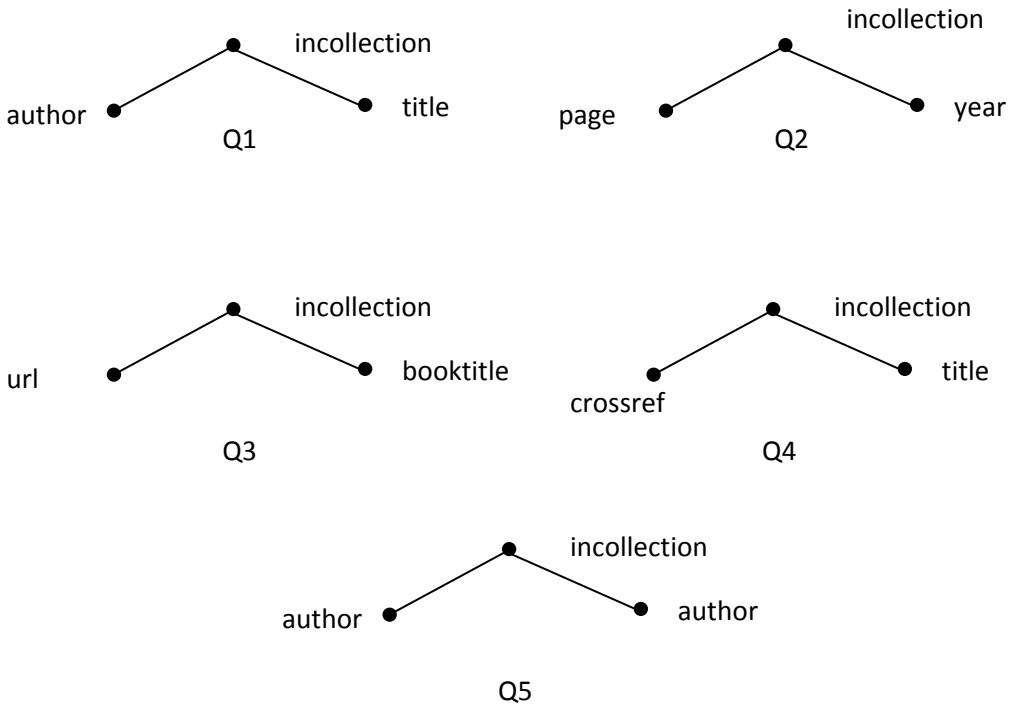
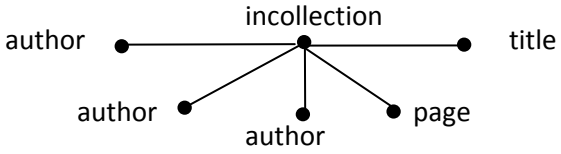
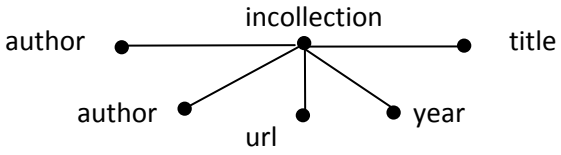


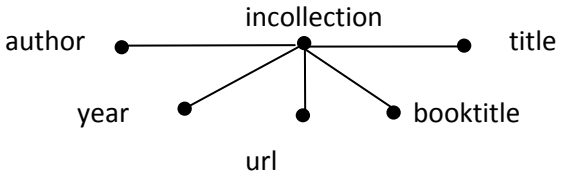
Figure 4.7 Query of small size



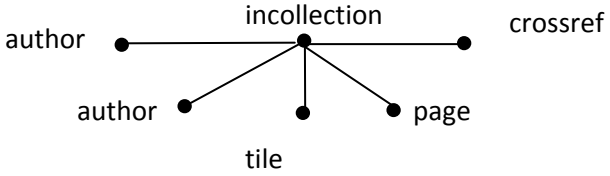
Q6



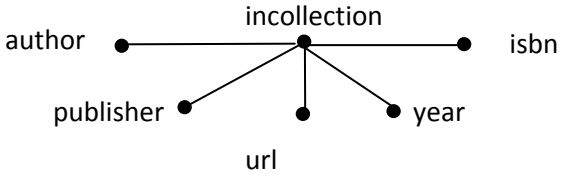
Q7



Q8

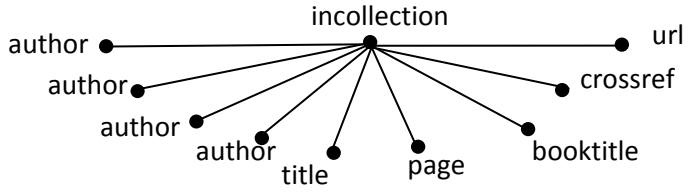


Q9

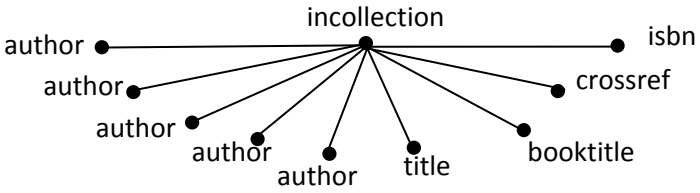


Q10

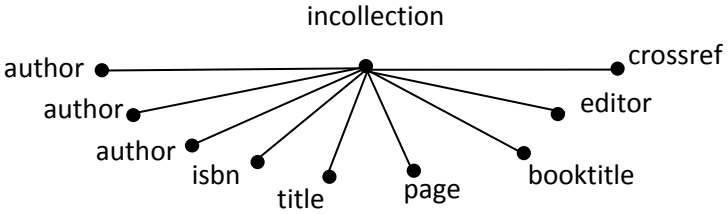
Figure 4.8 Query of median size



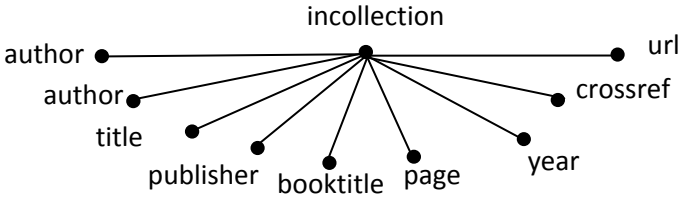
Q11



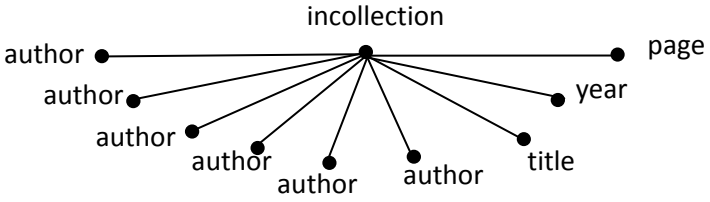
Q12



Q13



Q14



Q15

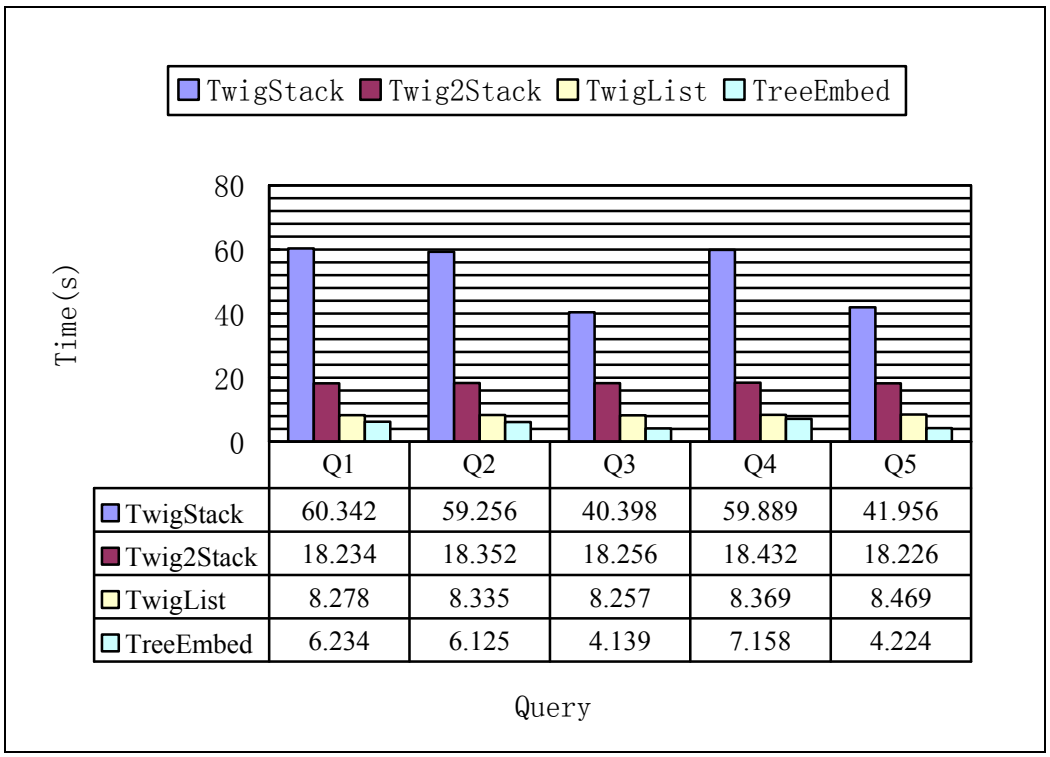
Figure 4.9 Query of large size

4.5.2 Test results

Figure 4.10 to 4.12 show the results of this experiment. From these charts, we see that the ***TreeEmbed*** method beats all the other methods in this test. When the queries are small, ***TreeEmbed*** finishes most of them within 7 seconds, whereas ***TwigStack*** takes about 1 minute to finish a query. The performance of ***Twig^{2}Stack*** is quite better than *TwigSatck* but still takes around 18 seconds. Only ***TwigList*** is close to ***TreeEmbed***, which is around 8 seconds. For the median size queries, most of them can be finished in 12 seconds by ***TreeEmbed***, slightly longer than small size queries. However, the query time of ***TwigStack*** and ***Twig^{2}Stack*** increase rapidly, nearly doubled than small size test results. The average difference between ***TwigList*** and ***TreeEmbed*** increases to more than 18 seconds. For the Large size queries, the average query time of ***TreeEmbed*** is around 40 seconds, but it still performed best in this group. Especially, the average difference between ***TwigList*** and ***TreeEmbed*** becames 60 seconds. ***TwigStack*** has to spend about 8 minutes to finish a query. ***Twig^{2}Stack*** is much better, but needs more than 2 minutes.

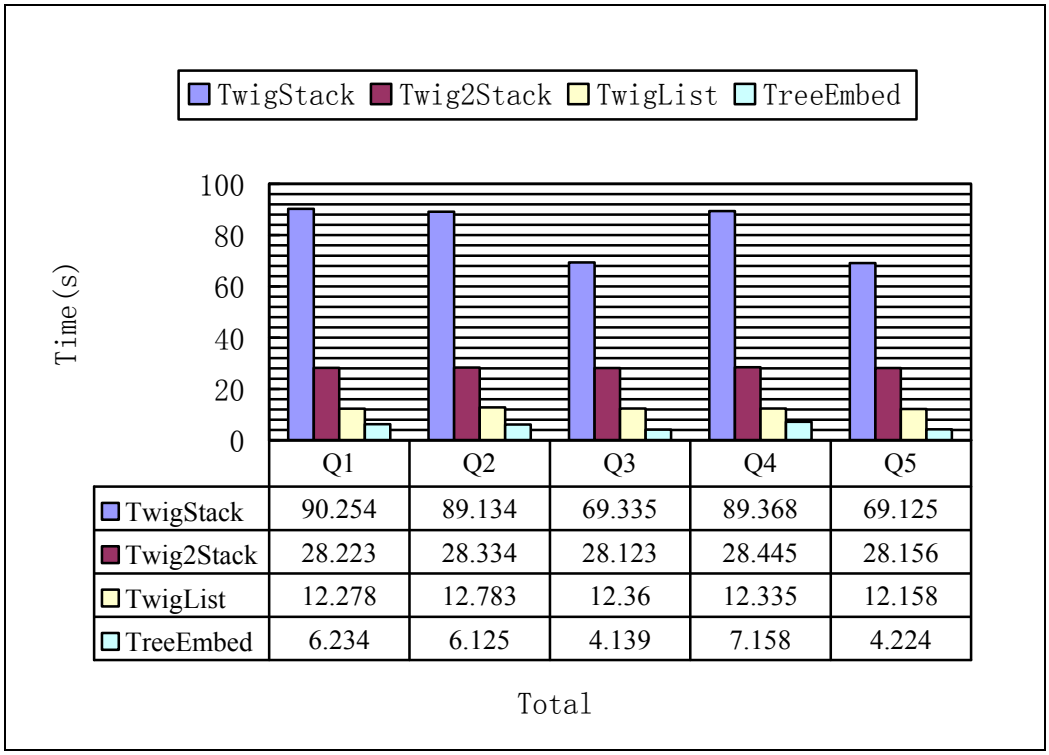
We find that the performance of ***TwigSatck*** and ***TreeEmbed*** are less stable than ***Twig^{2}Stack*** and ***TwigList***, especially for the group of the small size queries (e.g. Q3, Q5 and Q10), where their performance variations are more radical than those of large queries. Due to the randomness of ***XB-tree***, the ***TreeEmbed*** method may skip some leaf nodes. As the conclusion mentioned in [4], for data sets with solutions concentrated around certain portions of the data, the impact of ***XB-trees*** is more significant since many internal nodes can be skipped. This can be an explanation for the randomness of the test result.

Overall, *TreeEmbed* works best in this experiment. The results show that by using ***XB-tree***, ***TreeEmbed*** can immediately jump to potential matching documents without running through each document one by one to carry out an embedding checking.



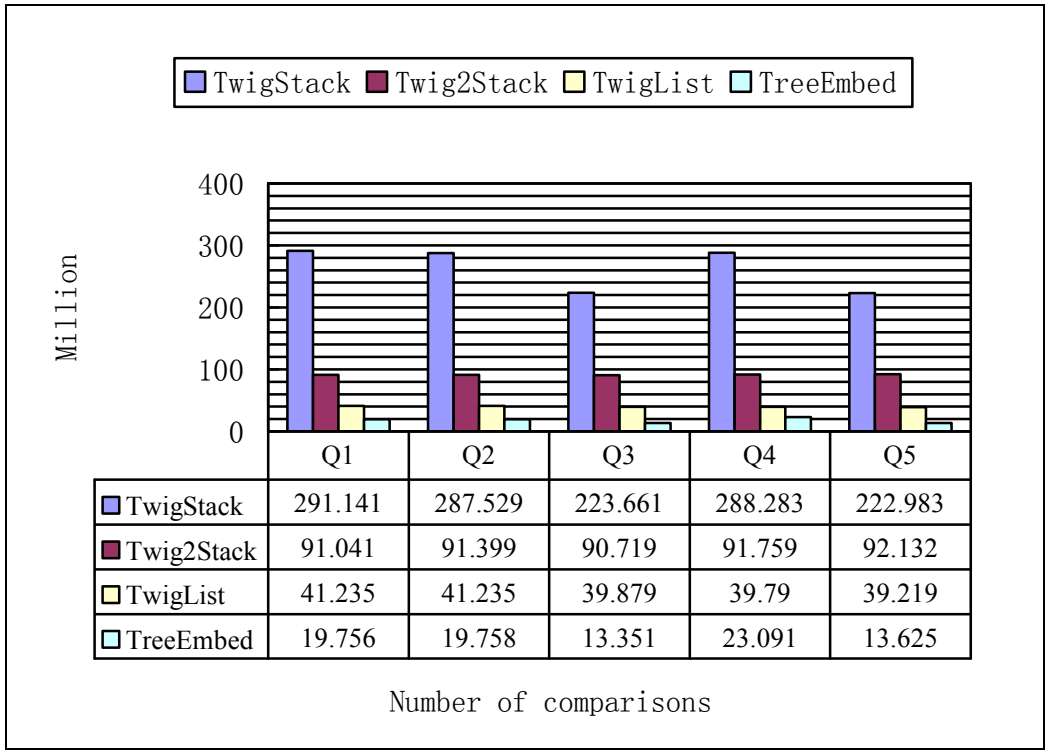
	Q1	Q2	Q3	Q4	Q5
TwigStack	60.342	59.256	40.398	59.889	41.956
Twig2Stack	18.234	18.352	18.256	18.432	18.226
TwigList	8.278	8.335	8.257	8.369	8.469
TreeEmbed	6.234	6.125	4.139	7.158	4.224

Figure 4.10 (a) Query Time in Group One



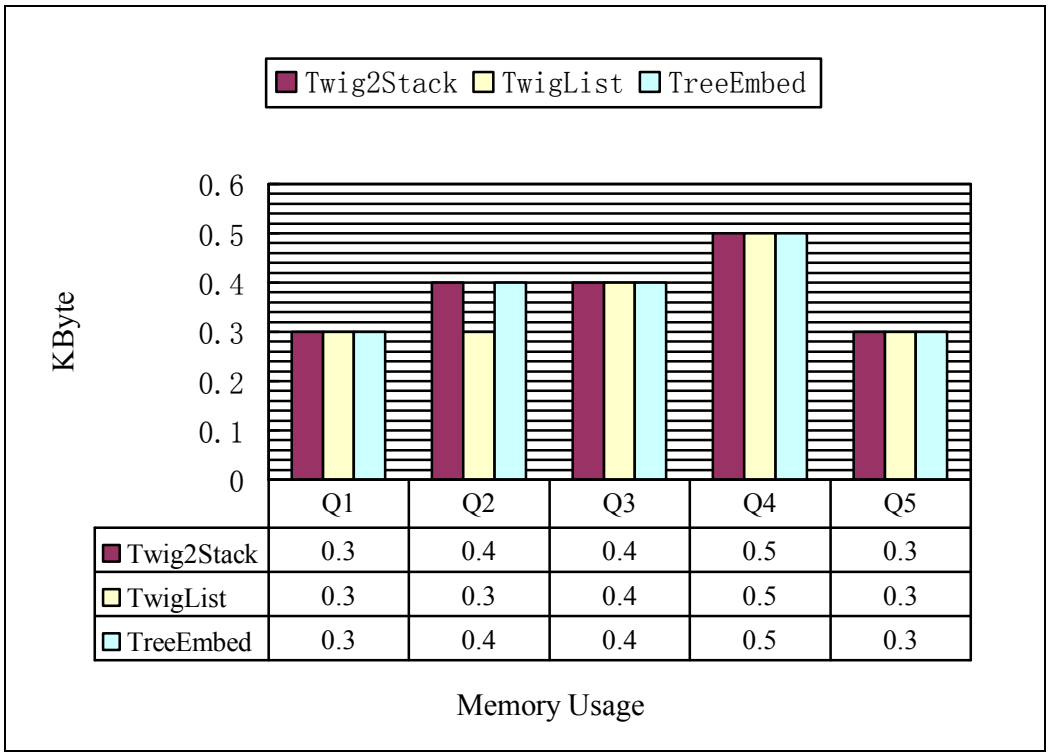
	Q1	Q2	Q3	Q4	Q5
TwigStack	90.254	89.134	69.335	89.368	69.125
Twig2Stack	28.223	28.334	28.123	28.445	28.156
TwigList	12.278	12.783	12.36	12.335	12.158
TreeEmbed	6.234	6.125	4.139	7.158	4.224

Figure 4.10 (b) Total Execution Time of Group One



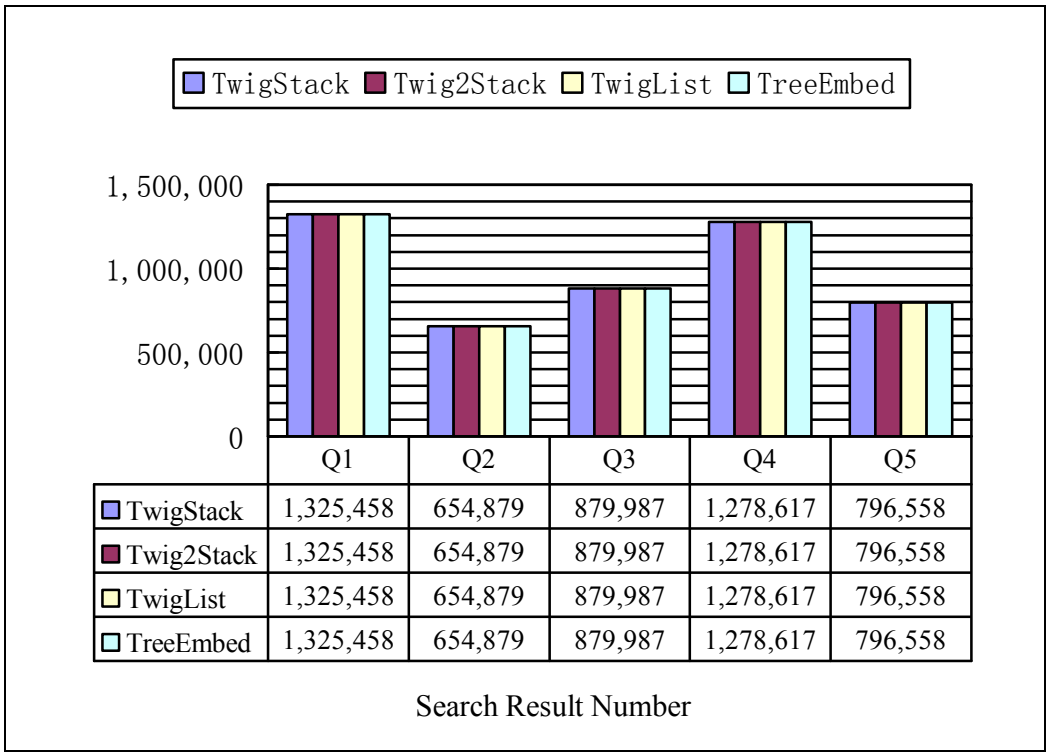
	Q1	Q2	Q3	Q4	Q5
TwigStack	291.141	287.529	223.661	288.283	222.983
Twig2Stack	91.041	91.399	90.719	91.759	92.132
TwigList	41.235	41.235	39.879	39.79	39.219
TreeEmbed	19.756	19.758	13.351	23.091	13.625

Figure 4.10 (c) Total number of comparisons in Group One



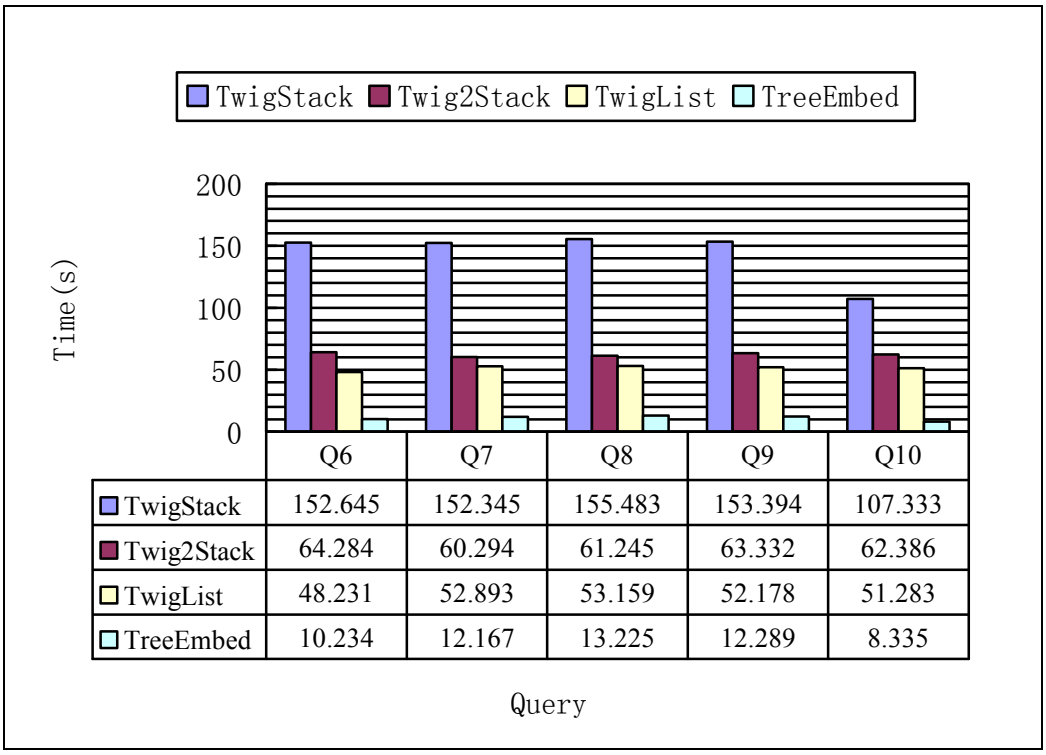
	Q1	Q2	Q3	Q4	Q5
Twig2Stack	0.3	0.4	0.4	0.5	0.3
TwigList	0.3	0.3	0.4	0.5	0.3
TreeEmbed	0.3	0.4	0.4	0.5	0.3

Figure 4.10 (d) Memory Usage in Group One



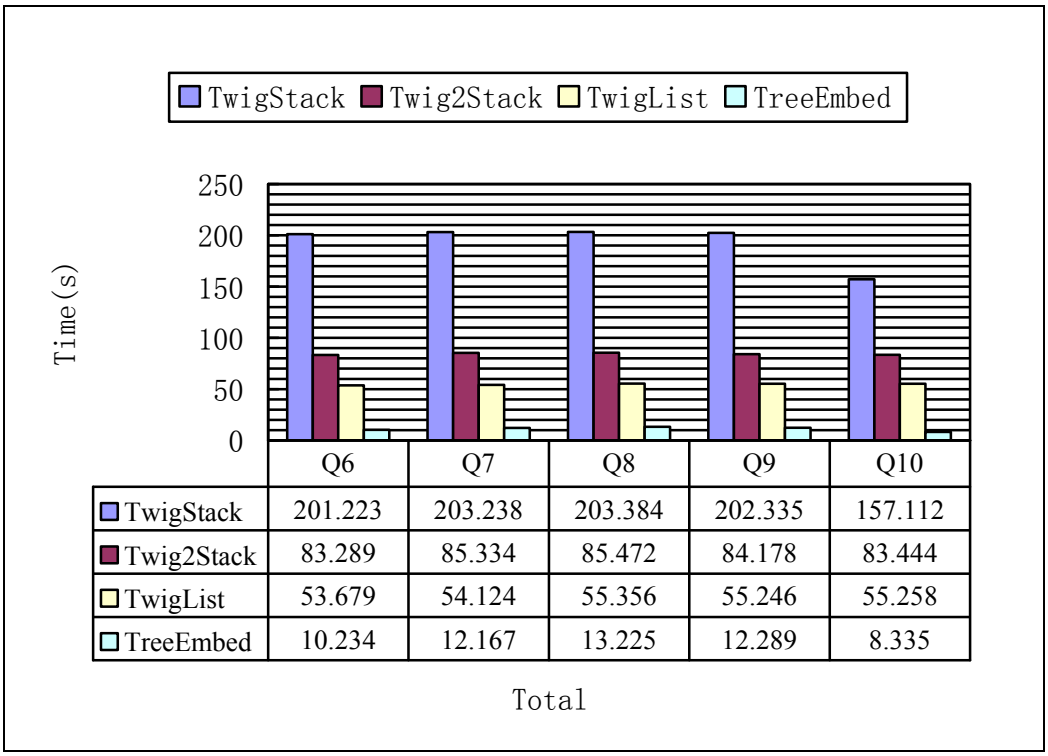
	Q1	Q2	Q3	Q4	Q5
TwigStack	1,325,458	654,879	879,987	1,278,617	796,558
Twig2Stack	1,325,458	654,879	879,987	1,278,617	796,558
TwigList	1,325,458	654,879	879,987	1,278,617	796,558
TreeEmbed	1,325,458	654,879	879,987	1,278,617	796,558

Figure 4.10 (e) Number of Search Results in Group One



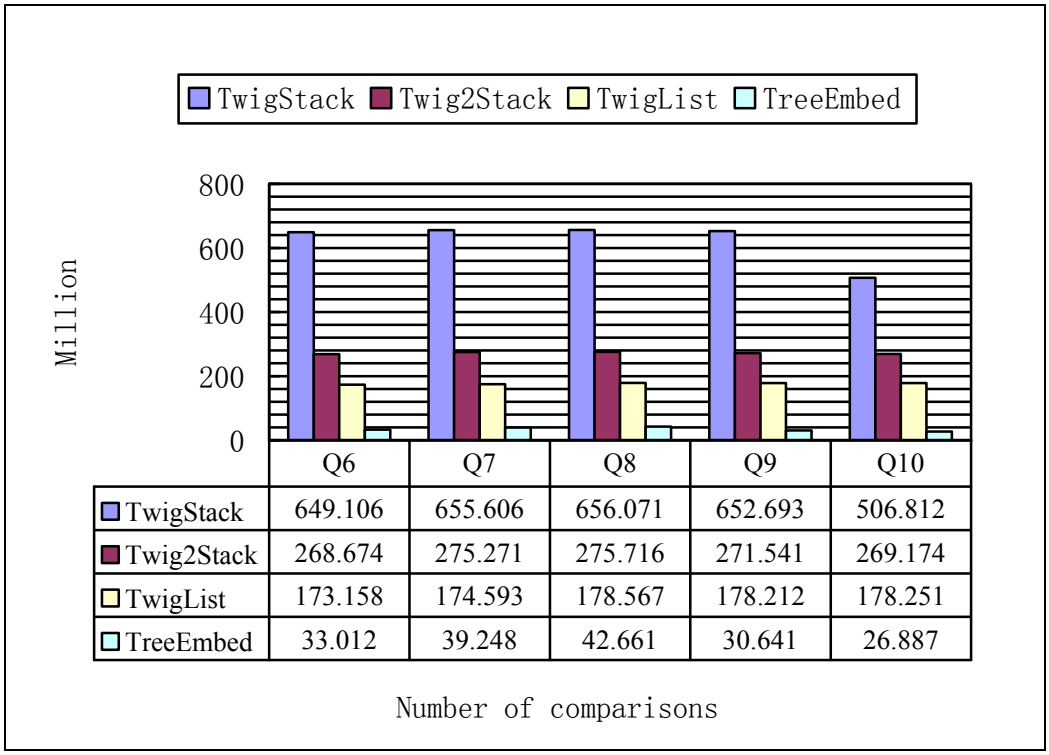
	Q6	Q7	Q8	Q9	Q10
TwigStack	152.645	152.345	155.483	153.394	107.333
Twig2Stack	64.284	60.294	61.245	63.332	62.386
TwigList	48.231	52.893	53.159	52.178	51.283
TreeEmbed	10.234	12.167	13.225	12.289	8.335

Figure 4.11 (a) Query time in Group Two



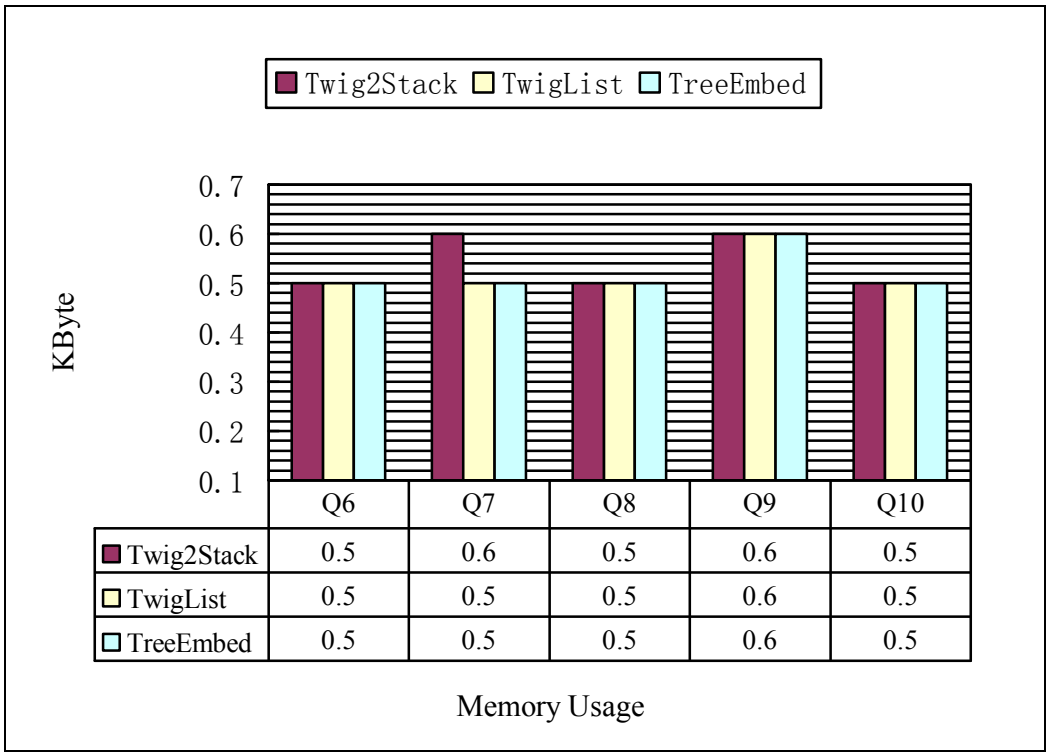
	Q6	Q7	Q8	Q9	Q10
TwigStack	201.223	203.238	203.384	202.335	157.112
Twig2Stack	83.289	85.334	85.472	84.178	83.444
TwigList	53.679	54.124	55.356	55.246	55.258
TreeEmbed	10.234	12.167	13.225	12.289	8.335

Figure 4.11 (b) Total Execution Time of Group Two



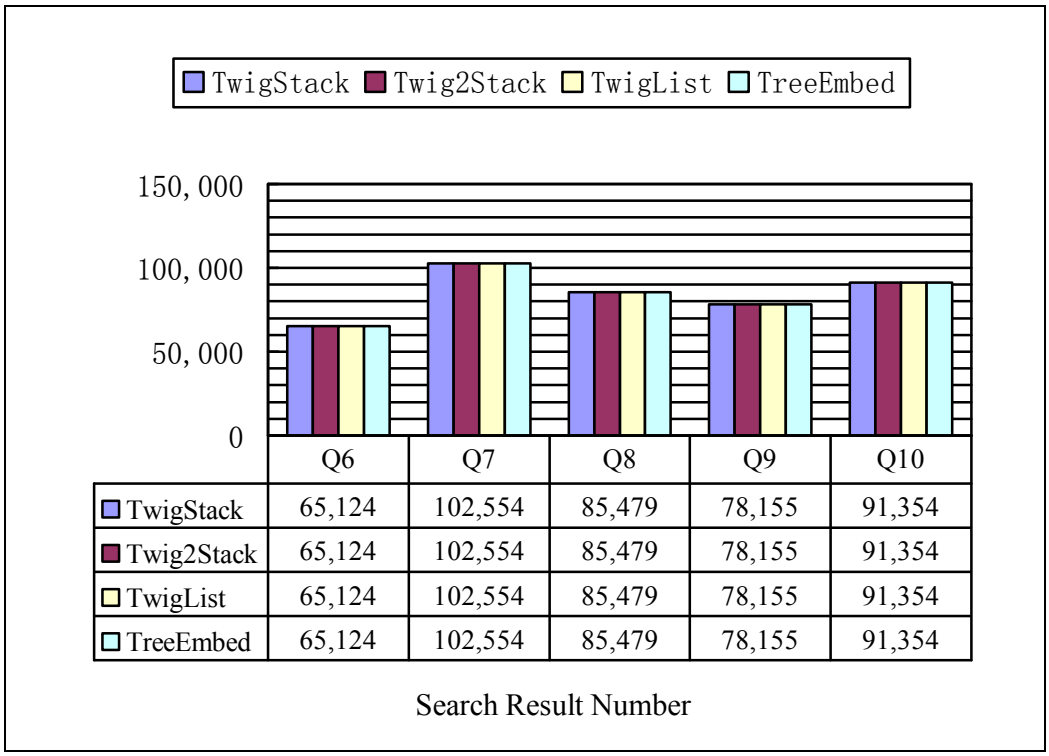
	Q6	Q7	Q8	Q9	Q10
TwigStack	649.106	655.606	656.071	652.693	506.812
Twig2Stack	268.674	275.271	275.716	271.541	269.174
TwigList	173.158	174.593	178.567	178.212	178.251
TreeEmbed	33.012	39.248	42.661	30.641	26.887

Figure 4.11 (c) Total number of comparisons in Group Two



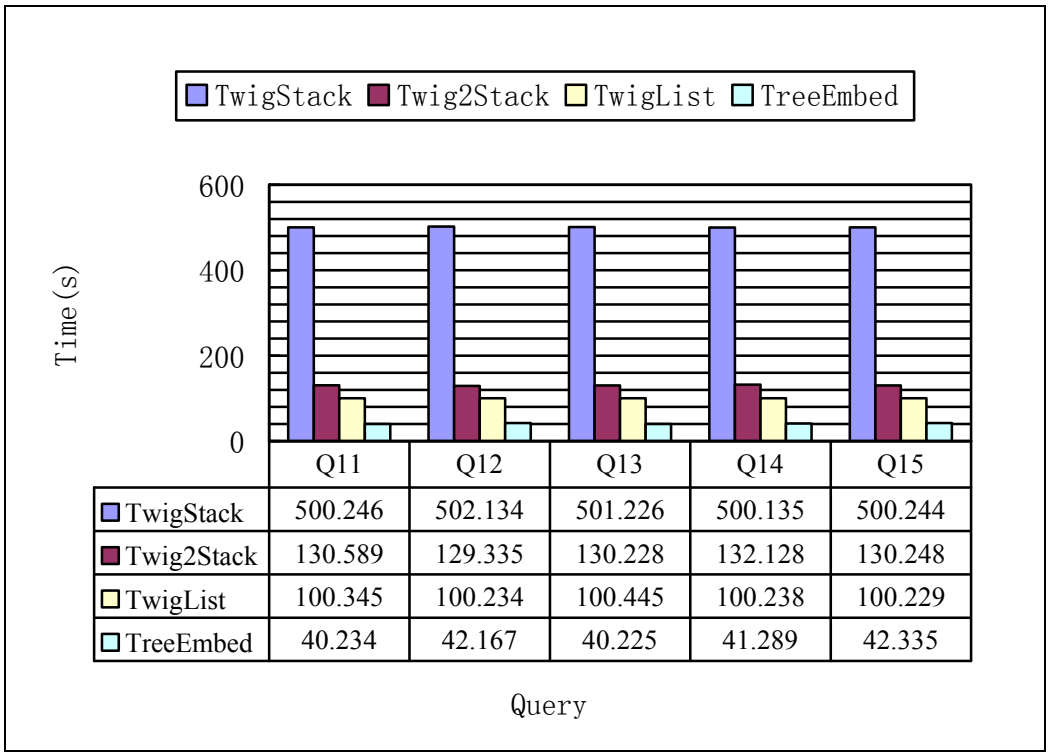
	Q6	Q7	Q8	Q9	Q10
Twig2Stack	0.5	0.6	0.5	0.6	0.5
TwigList	0.5	0.5	0.5	0.6	0.5
TreeEmbed	0.5	0.5	0.5	0.6	0.5

Figure 4.11 (d) Memory Usage in Group Two



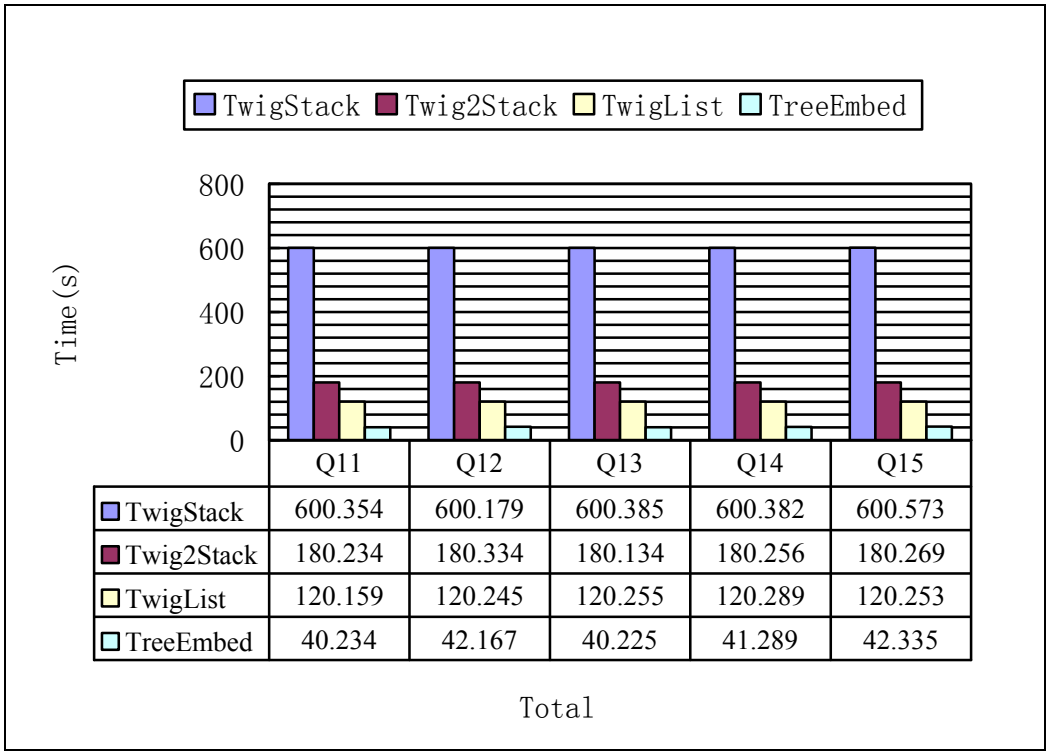
	Q6	Q7	Q8	Q9	Q10
TwigStack	65,124	102,554	85,479	78,155	91,354
Twig2Stack	65,124	102,554	85,479	78,155	91,354
TwigList	65,124	102,554	85,479	78,155	91,354
TreeEmbed	65,124	102,554	85,479	78,155	91,354

Figure 4.11 (e) Number of Search Results in Group Two



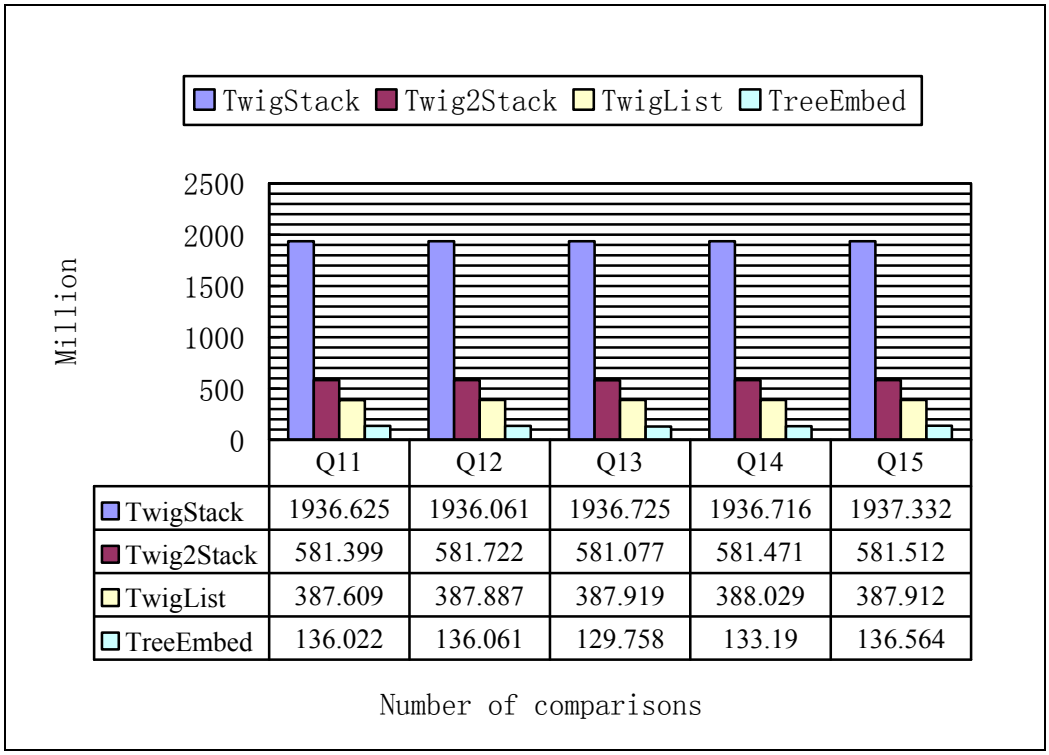
	Q11	Q12	Q13	Q14	Q15
TwigStack	500.246	502.134	501.226	500.135	500.244
Twig2Stack	130.589	129.335	130.228	132.128	130.248
TwigList	100.345	100.234	100.445	100.238	100.229
TreeEmbed	40.234	42.167	40.225	41.289	42.335

Figure 4.12 (a) Query Time in Group Three



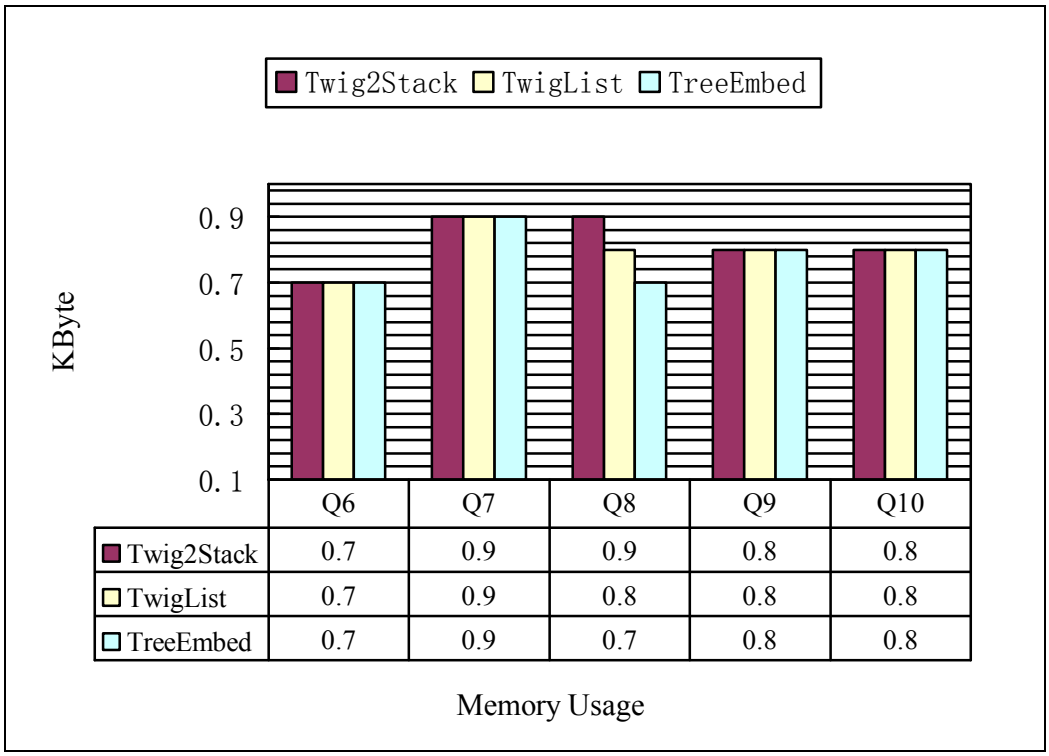
	Q11	Q12	Q13	Q14	Q15
TwigStack	600.354	600.179	600.385	600.382	600.573
Twig2Stack	180.234	180.334	180.134	180.256	180.269
TwigList	120.159	120.245	120.255	120.289	120.253
TreeEmbed	40.234	42.167	40.225	41.289	42.335

Figure 4.12 (b) Total Execution time in Group Three



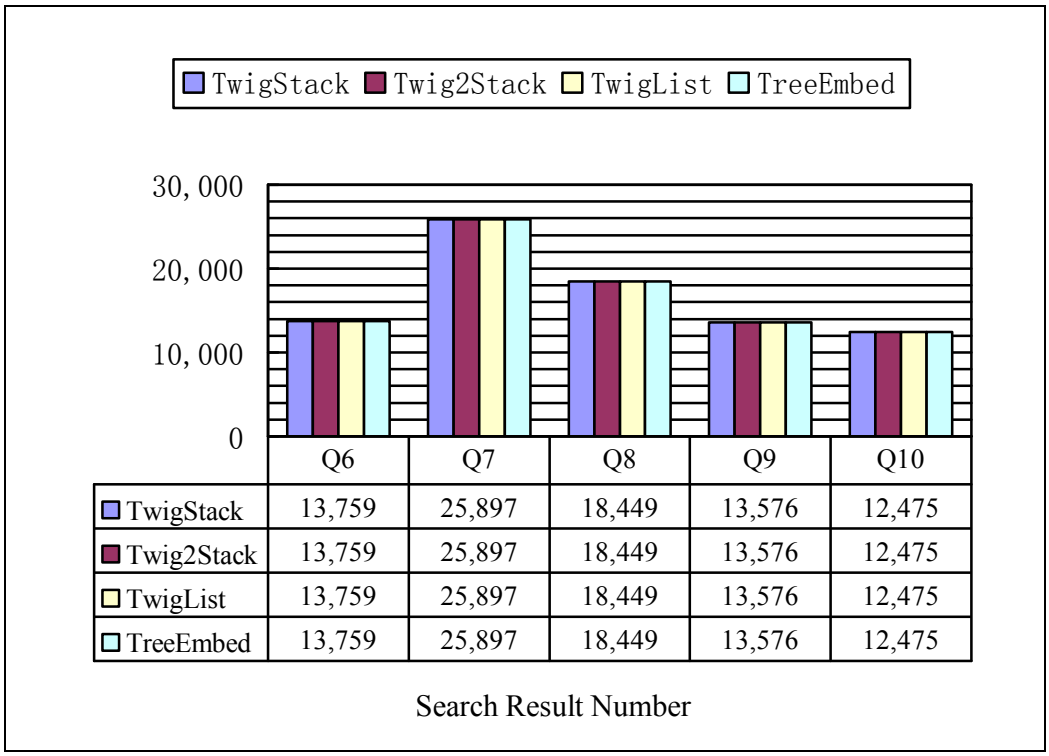
	Q11	Q12	Q13	Q14	Q15
TwigStack	1936.625	1936.061	1936.725	1936.716	1937.332
Twig2Stack	581.399	581.722	581.077	581.471	581.512
TwigList	387.609	387.887	387.919	388.029	387.912
TreeEmbed	136.022	136.061	129.758	133.19	136.564

Figure 4.12 (c) Total number of comparisons in Group Three



	Q6	Q7	Q8	Q9	Q10
Twig2Stack	0.7	0.9	0.9	0.8	0.8
TwigList	0.7	0.9	0.8	0.8	0.8
TreeEmbed	0.7	0.9	0.7	0.8	0.8

Figure 4.12 (d) Memory Usage in Group Three



	Q6	Q7	Q8	Q9	Q10
TwigStack	13,759	25,897	18,449	13,576	12,475
Twig2Stack	13,759	25,897	18,449	13,576	12,475
TwigList	13,759	25,897	18,449	13,576	12,475
TreeEmbed	13,759	25,897	18,449	13,576	12,475

Figure 4.12 (e) Number of Search Results in Group Three

4.6 Experiments on XMark

In this experiment, we will make tests on XMark data set. The data set structure was shown in Figure 4.1 (c). We use 5 factors to test the scalability of the 4 algorithms.

4.6.1 Queries

In this test, we used only two queries. One is a simple path and the other is of a tree structure, as shown in Table 4.7.

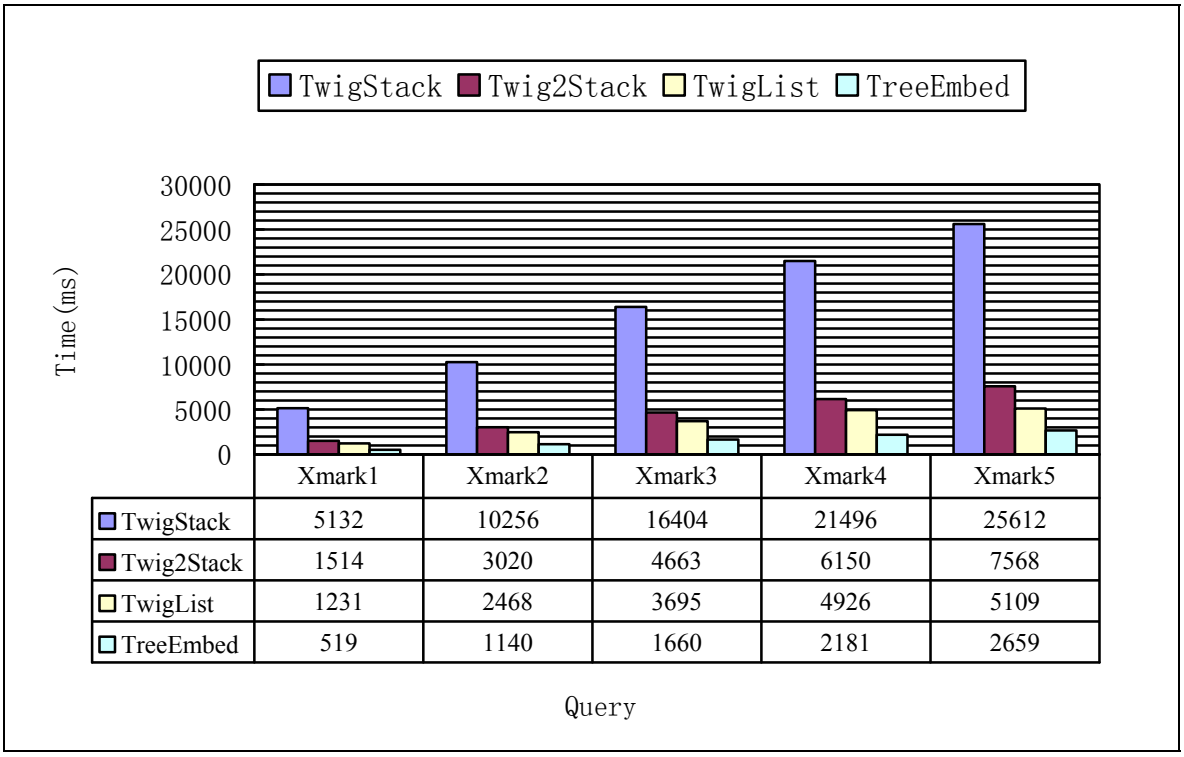
Table 4.7 The queries of XMark

Name	Query Trees
Q1	//item[description]//mail
Q2	//open_auction[.//annotation[.//person]//parlist]//bidder//increase

4.6.2 Results

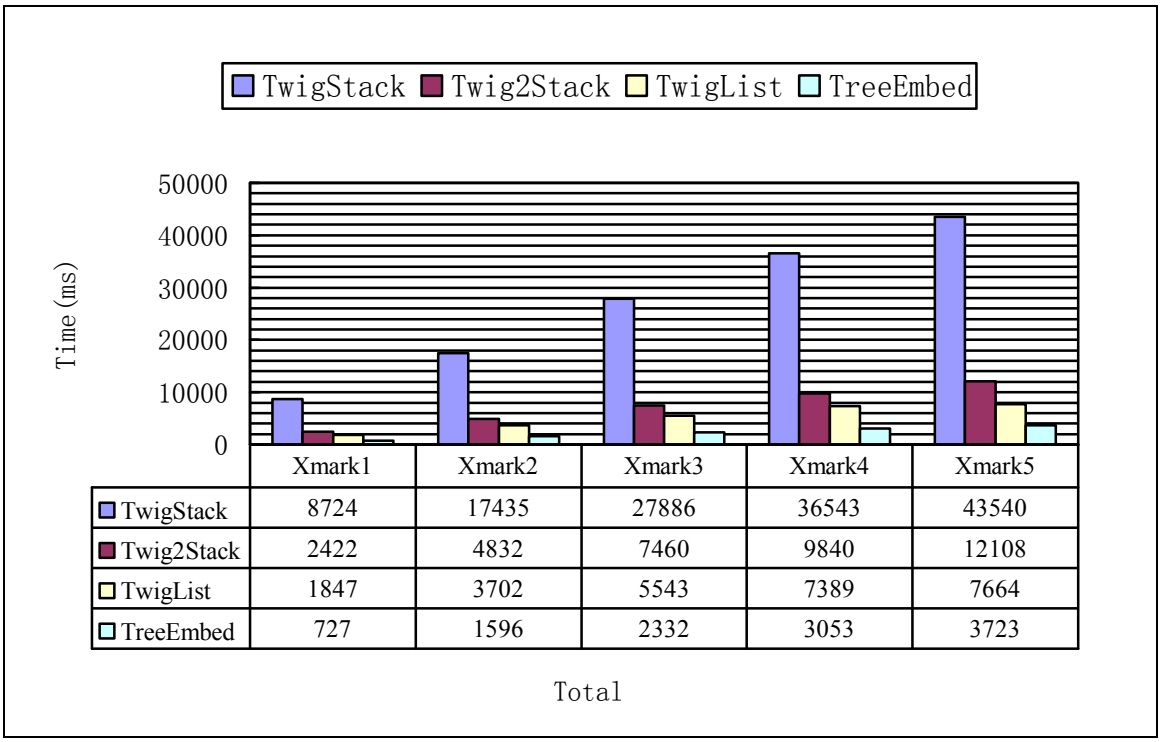
Figure 4.13 show the test results on XMark. The data size of this test can be found in Table 4.1. We vary the XMark scale factor from 1 to 5. From the charts in Figure 4.13, we see that the times of all the four algorithms grow linearly in the document sizes. Again, we see that the ***TreeEmbed*** method beat the other methods in this experiment. Something needs to notice is that as the size of queries increases the times spent by ***TwigStack*** and ***Twig^{2}Stack*** grow much faster than ***TwigList*** and ***TreeEmbed***. For Q2, the query time of ***TwigStack*** and ***Twig^{2}Stack*** is nearly 4 times larger than that for Q1, while for ***TwigList*** and ***TreeEmbed*** the query time of Q2 is only around 2 times larger than that for Q1. The reason for this is that when ***TwigStack*** and ***Twig^{2}Stack*** use a join operation to enumerate results, a huge

volume of intermediate results will be produced and manipulated. But ***TwigList*** and ***TreeEmbed*** do not generate any intermediate results. So the impact of the query size is not so large as ***TwigStack*** and ***Twig^{2}Stack***.



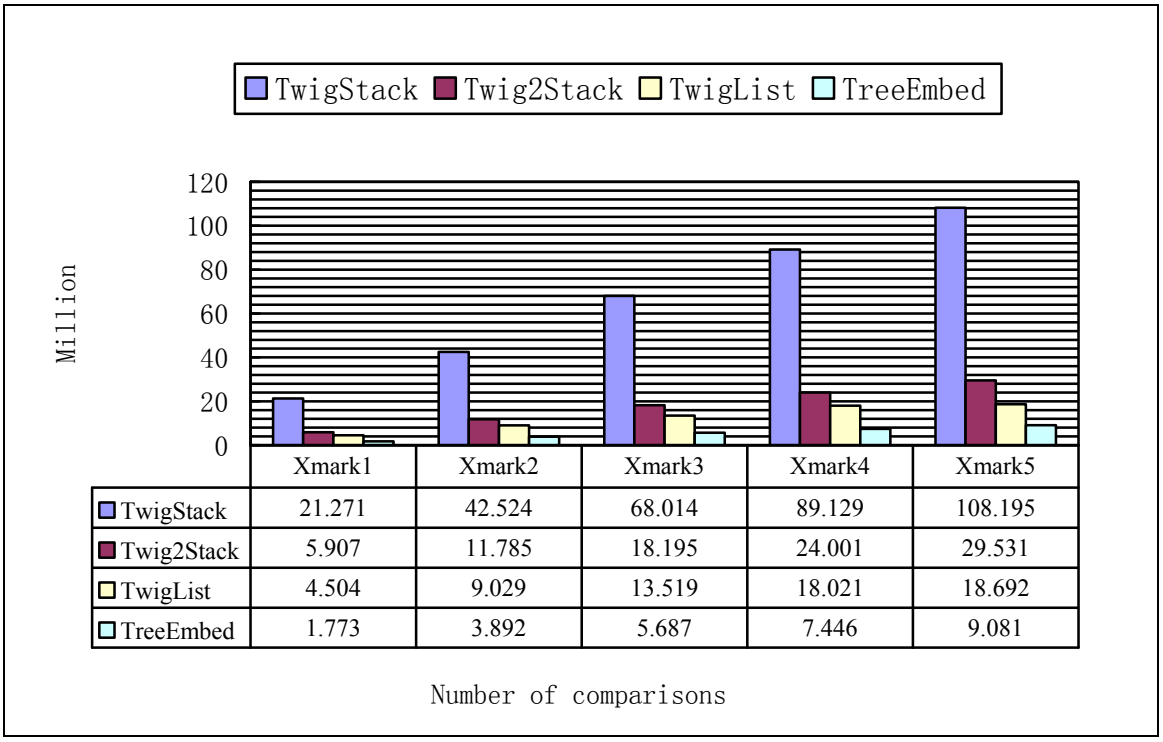
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	5132	10256	16404	21496	25612
Twig2Stack	1514	3020	4663	6150	7568
TwigList	1231	2468	3695	4926	5109
TreeEmbed	519	1140	1660	2181	2659

Figure 4.13 (a) The query time of XMark Q1



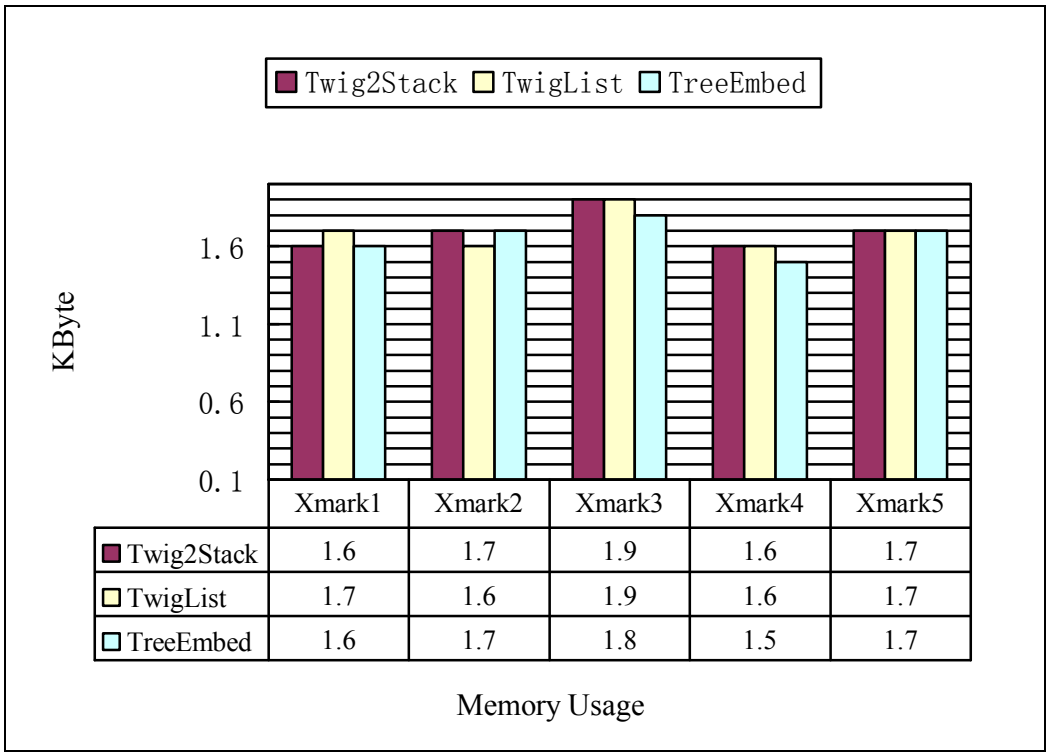
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	8724	17435	27886	36543	43540
Twig2Stack	2422	4832	7460	9840	12108
TwigList	1847	3702	5543	7389	7664
TreeEmbed	727	1596	2332	3053	3723

Figure 4.13 (b) Total Execution time in XMark Q1



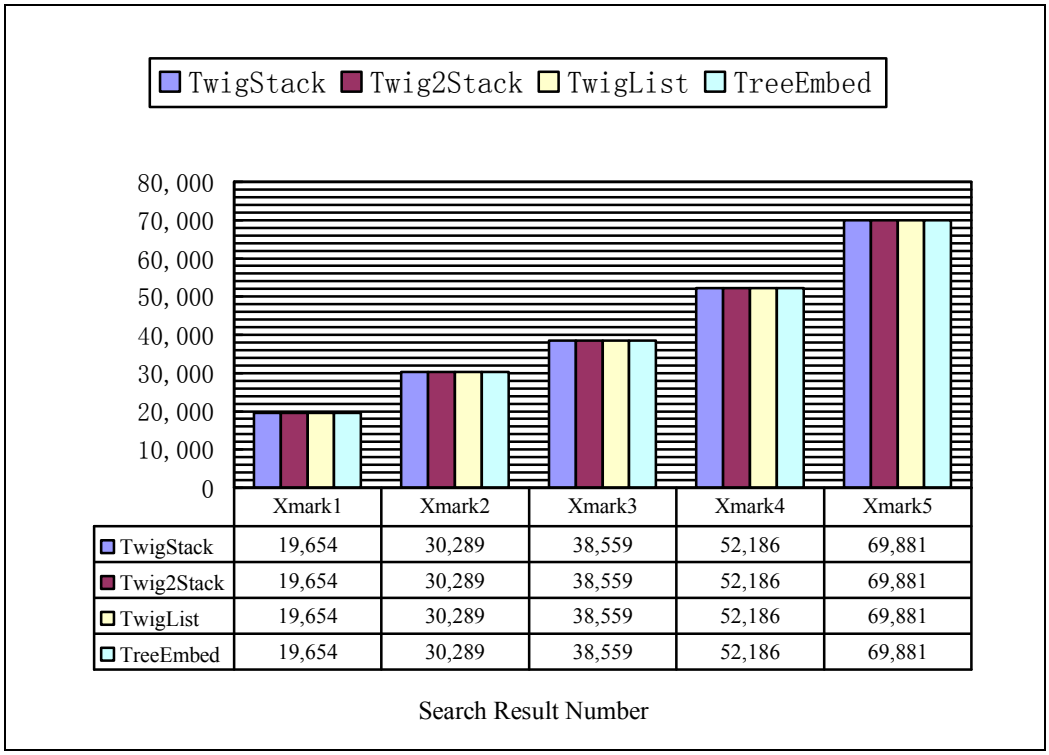
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	21.271	42.524	68.014	89.129	108.195
Twig2Stack	5.907	11.785	18.195	24.001	29.531
TwigList	4.504	9.029	13.519	18.021	18.692
TreeEmbed	1.773	3.892	5.687	7.446	9.081

Figure 4.13 (c) Total number of comparisons in XMark Q1



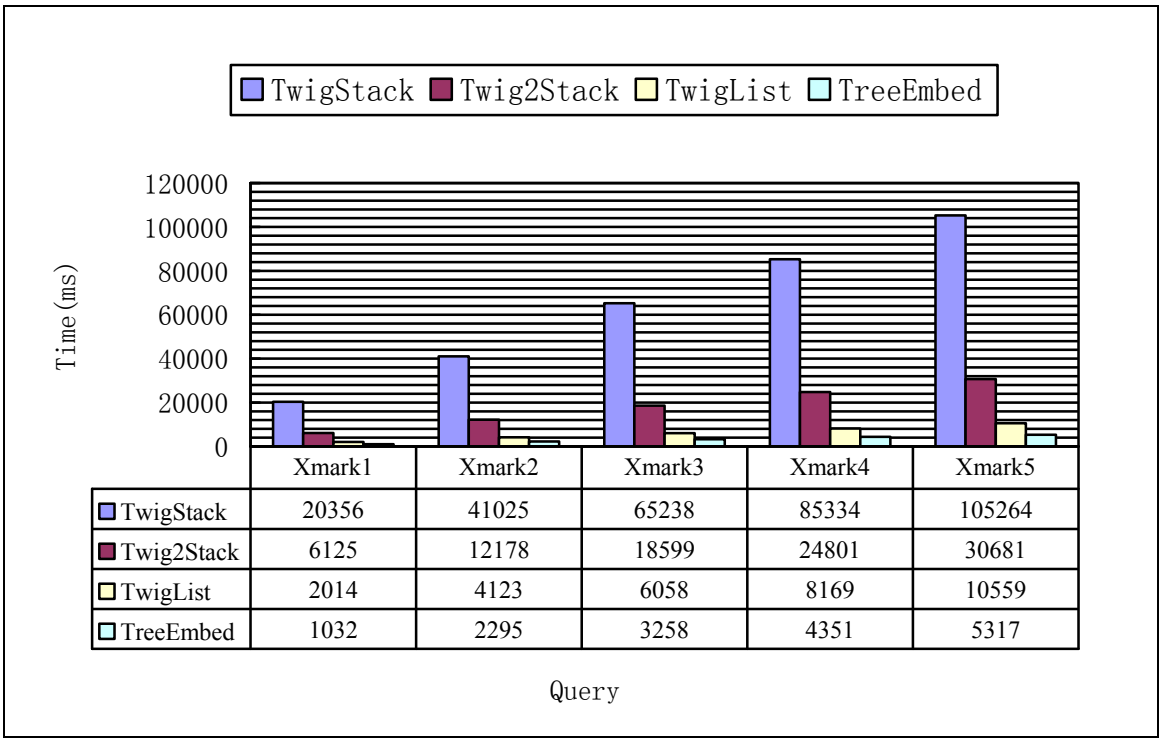
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
Twig2Stack	1.6	1.7	1.9	1.6	1.7
TwigList	1.7	1.6	1.9	1.6	1.7
TreeEmbed	1.6	1.7	1.8	1.5	1.7

Figure 4.12 (d) Memory Usage in XMark Q1



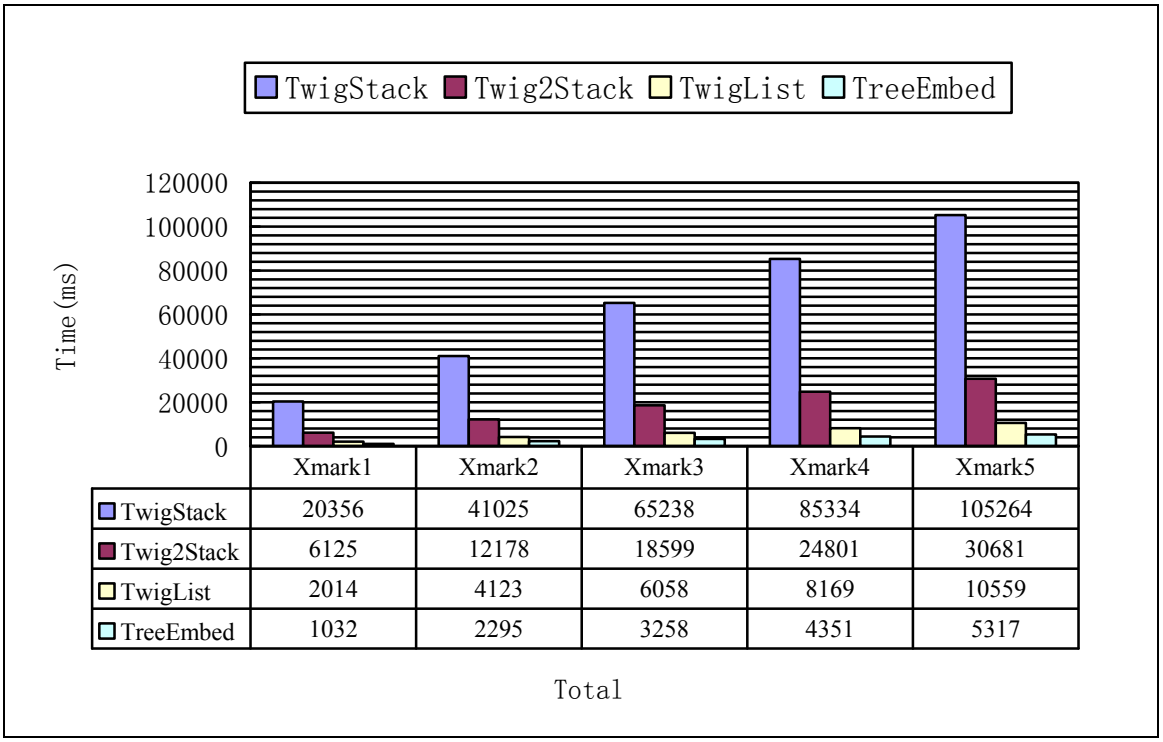
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	19,654	30,289	38,559	52,186	69,881
Twig2Stack	19,654	30,289	38,559	52,186	69,881
TwigList	19,654	30,289	38,559	52,186	69,881
TreeEmbed	19,654	30,289	38,559	52,186	69,881

Figure 4.12 (e) Number of Search Results in XMark Q1



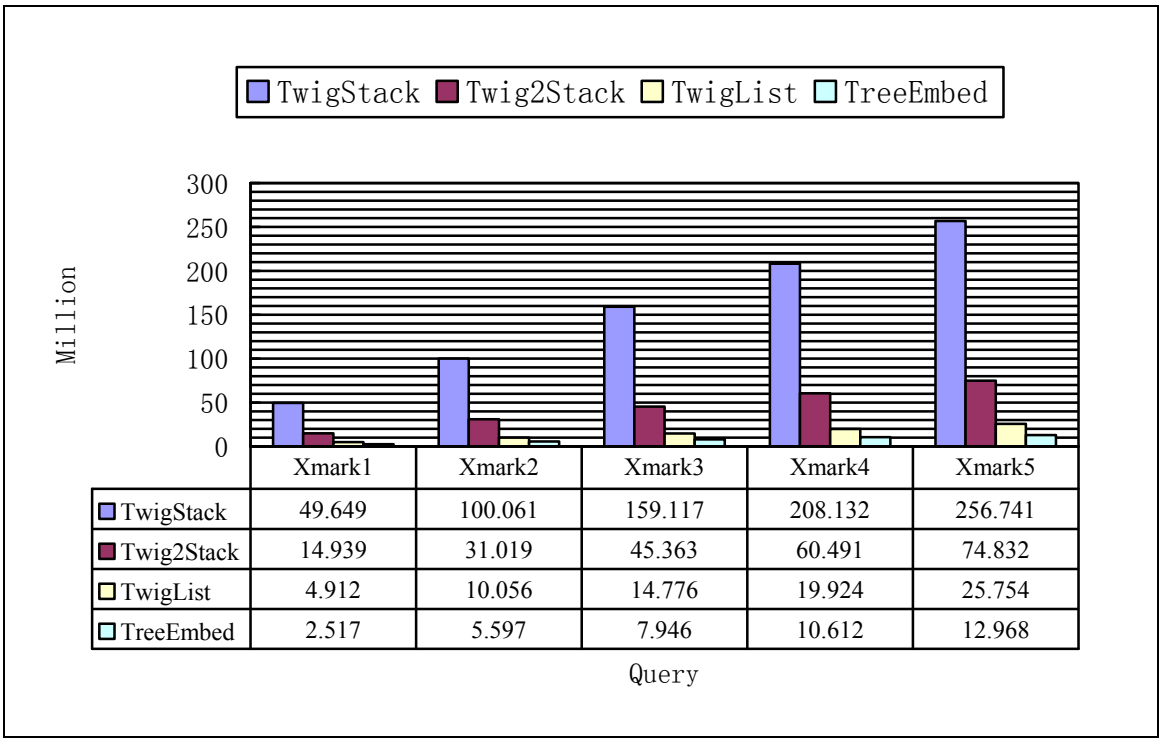
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	20356	41025	65238	85334	105264
Twig2Stack	6125	12178	18599	24801	30681
TwigList	2014	4123	6058	8169	10559
TreeEmbed	1032	2295	3258	4351	5317

Figure 4.14 (a) The query time in Xmark Q2



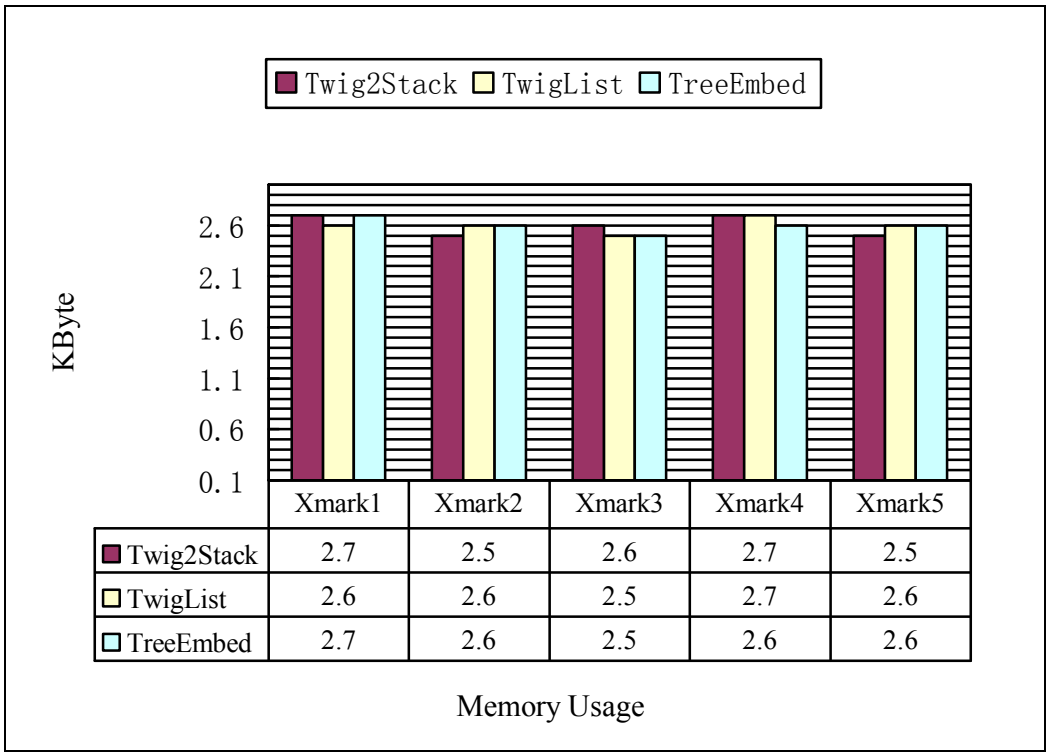
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	20356	41025	65238	85334	105264
Twig2Stack	6125	12178	18599	24801	30681
TwigList	2014	4123	6058	8169	10559
TreeEmbed	1032	2295	3258	4351	5317

Figure 4.14 (b) Total Execution time in XMark Q2



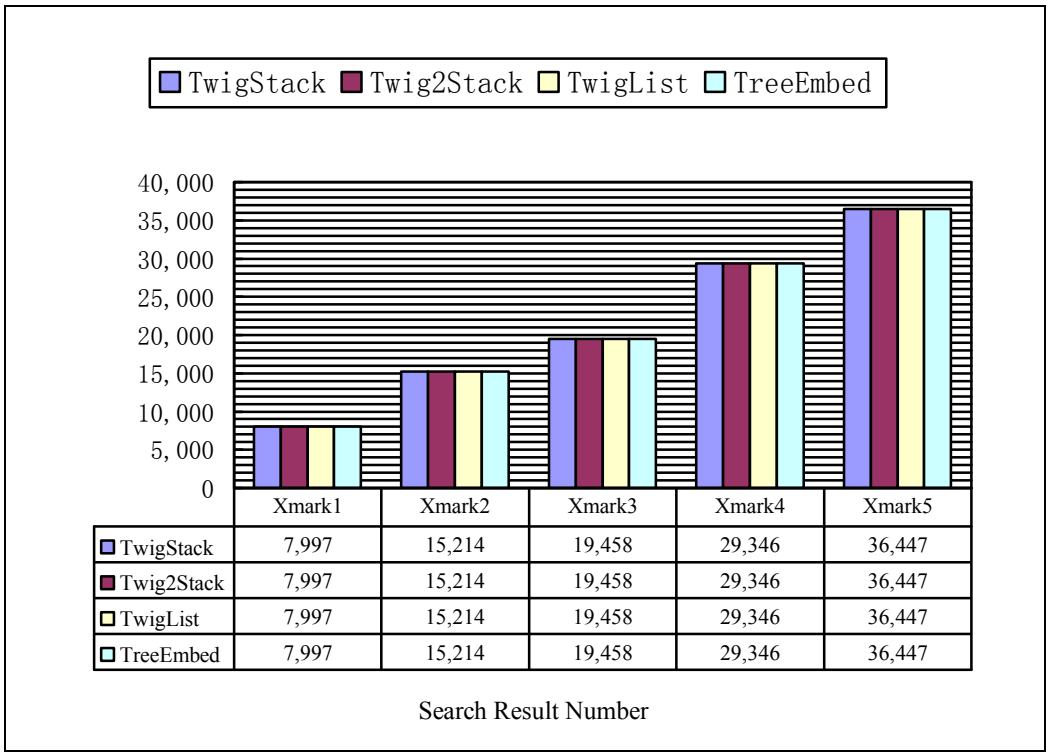
	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	49.649	100.061	159.117	208.132	256.741
Twig2Stack	14.939	31.019	45.363	60.491	74.832
TwigList	4.912	10.056	14.776	19.924	25.754
TreeEmbed	2.517	5.597	7.946	10.612	12.968

Figure 4.14 (c) Total number of comparisons in XMark Q2



	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
Twig2Stack	2.7	2.5	2.6	2.7	2.5
TwigList	2.6	2.6	2.5	2.7	2.6
TreeEmbed	2.7	2.6	2.5	2.6	2.6

Figure 4.12 (d) Memory Usage in XMark Q2



	Xmark1	Xmark2	Xmark3	Xmark4	Xmark5
TwigStack	7,997	15,214	19,458	29,346	36,447
Twig2Stack	7,997	15,214	19,458	29,346	36,447
TwigList	7,997	15,214	19,458	29,346	36,447
TreeEmbed	7,997	15,214	19,458	29,346	36,447

Figure 4.12 (e) Number of Search Results in XMark Q2

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we developed a new algorithm to evaluate tree pattern queries based on unordered tree matching. The motivation for this work is to overcome some limitations of current query evaluation methods, such as redundant computation of subtrees which contain the matching nodes, or reading and processing parts of the streams which cannot contain useful nodes. In order to achieve a better performance, we use a new labeling method for tree pattern queries, and treat the data set in a bottom up way. The core idea of the method is to reconstruct a tree from data streams, during which each node v that matches a query node will be inserted into the tree and associated with a query node stream $QS(v)$ such that for each node q in QS (v) $T[v]$ embeds $Q[q]$. Especially, the algorithm can be adapted into an indexing environment with ***XB-tree*** [4] being used.

An overview of the tree pattern matching problem for XML databases is presented, which provides some background information on the tree encoding, the data streams, as well as the ***XB-tree*** index technique to solve this problem. We have also surveyed the literature related to the tree pattern matching problem, and proposed a new bottom up query evaluation algorithm ***TreeEmbed***. In addition, how to combine ***TreeEmbed*** with the ***XB-tree*** index is discussed in great detail. We implemented the bottom up

tree reconstruction algorithm and the ***XB-tree*** index over the data streams, and compared the performance of our method with three other algorithms: ***TwigStack***, ***Twig^{2}Stack*** and ***TwigList***, which shows that our query evaluation method ***TreeEmbed*** is promising.

5.2 Future Work

As the future work, we will continue our research in the following aspects.

- ***XB-tree* enhance**

  The ***XB-tree*** as an index technique plays a very important role in our algorithm. We will do more test on its dynamical maintenance when the deletion and insertion of nodes are conducted.

- **Ordered and Unordered**

  Our algorithm only supports the unordered tree matching. We will make more analysis on our tree reconstruction process to find a way to support the ordered tree matching. [3]

- **Practical Example**

  In our experiments, our algorithm shows a high efficiency. But more work has to be done to make it useful in practice, especially, to extend it to do the image search, the protein sequence search and the social network search, etc. Also, a graphical interface needs to be established.

- **Support XQuery**

  XQuery is a new query language designed specifically for querying XML data. Its current version is 1.0 [14] and it will become a standard

for processing XML data sets. So, in the near future, any query will be submitted in this standard format. One of our next main tasks is to integrate our algorithm into this language to speed up the query evaluation.

Reference

[1] J. Li and J. Wang. Fast matching of twig patterns. *In Proc. DEXA*, 2008.

[2] [C.chung, J.Min, and K.Shim, APEX: An adaptive path index for XML data, *ACM SIGMOD*, Jun 2002.

[3] Pekka Kilpeläinen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAMJ. Comput.*, 24(2):340-356, 1995

[4] N. Bruno, N. Koudas, and D. Srivastava, Holistic twig joins: Optimal XML pattern matching. *In Proc. SIGMOD.*, 2002.

[5] Yanjun Chen, A Time Optimal Algorithm for Evaluating Tree Pattern, *In SAC'.*, 2010

[6] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. *In Proc. ICDE.*, 2002.

[7] S. Chen, H-G. Li, J. Tatemura, W-P. Hsiung, D. Agrawa, and K.S. Candan, Twig^{2}Stack: Bottom-up Processing of Generalized Tree-Pattern Queries over XML Documents, *In Proc VLDB, Seoul, Korea,* Sept. 2006, pp. 283-294.

[8] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. *In Proc. DEXA*, 2003.

[9] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *In SIGMOD Rec.*, 2001.

[10] L. Qin, J. X. Yu, and B. Ding. TwigList: Make twig pattern matching fast. *In Proc. DASFAA*, 2007.

[11] U. of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/

[12] A.R.S. et al. The XML Benchmark Project Technical Report, http://www.xml-benchmark.org/ 2009

[13] M. Ley. Computer Science Bibliography. http://www.informatik.uni-trier.de/~ley/db/ 2010

[14] W3C Recommendation, XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/ 2007

[15] W3C Recommendation, XPath 1.0 XML Path Language (XPath)Version 1.0 http://www.w3.org/TR/xpath/ 1999

[16] Z. Jiang, C. Luo, W.-C. Hou, and Q. Z. D. Che. Efficient processing of XML twig pattern: A novel one-phase holistic solution. *In Proc. DEXA*, 2007.

[17] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-tree: Indexing XML data for efficient structural joins. *In Proc. ICDE*, 2003

[18] Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. *In Proc. SIGMOD*, 2002.

[19] T. Chen, J. Lu, and T.W. Ling, On Boosting Holism in XML Twig Pattern Matching, *In Proc. SIGMOD*, pp. 455- 466, 2005

[20] Nils Grimsmo and Truls A.Bjorklund, Towards Unifying Advances in Twig Join Algorithms, *In the Twenty-First Australasian Database Conference(ADC2010)*, Brisbane, Australia., 2010.