TJP: A Modified Twig Join Algorithm Based on the Pri-order Labeling Scheme

Ren JiaDong† and Yue LiWen†

† College of Information Science and Engineering, Yanshan University Qinhuangdao, Hebei, 066004 China

Summary
XML exploits a tree-structured data model for representing data, and XML queries specify patterns of selection predicates on multiple elements related by a tree structure. Finding all occurrences of such a twig pattern in an XML database is a core operation for XML query processing. A lot of algorithms have been proposed to process to XML twig pattern query based-on region labeling scheme, which can not support the updating of XML documents. In this paper, a new twig join algorithm TJP is proposed, for matching an XML query twig pattern, it is modification of *TwigStackList* based on a new labeling scheme. This labeling scheme is optimal to determine the relationships between nodes and can support efficiently dynamic updating of documents. TJP uses a list to cache some elements in input data streams; the length of list is not longer than that of the longest path in the XML documents. It is important technique for lists of branching nodes; we preserve the elements in list only when they contribute to the final query results. The algorithm is I/O and CPU optimal for queries with ancestor-descendant edge. When the twig pattern contains parent-child relationships below branching nodes, the algorithm TJP will get a much smaller set of intermediate results than previous join algorithms.

Key words:
XML, twig pattern, labeling scheme, join algorithm.

1. Introduction

With the development of XML, it has become a common standard for data representation and information exchange over the Internet. Although XML documents could have rather complex internal structures, they can generally be modeled as labeled and ordered trees. Most of researches focus on query processing over XML data that conformed to a tree-structured data model. In most XML query languages [1, 2], queries on XML data are commonly expressed in the form of twig pattern (a small tree).A twig pattern can be represented as a node-labeled tree whose edges are either parent-child or ancestor-descendant relationship.

Efficiently finding all occurrences of a twig pattern in a database is a core operation in XML query processing, both in relational implementations of XML databases and in native XML databases. In the past several years, many algorithms [3 -7] have been proposed. Most of them decomposed the twig pattern into a set of binary (parent-child and ancestor-descendant) relationships between pairs of nodes, the query twig pattern can be matched by (1) matching each of the binary structural relationships against the XML databases, and (2) joining together these basic matches. Zhang et al. [8] and Al-Khalifa et al. [4] proposed the merge join algorithms: MPMGJN 、 tree-merge and Stack- tree respectively, to match the binary relationships, and finally join together basic matches to get the final results. The limitation of these algorithms for matching query twig patterns is that intermediate result size get very large, even we control the size of input and output. To overcome this disadvantage, Bruno et al. [3] and Lu et al. [9] proposed holistic twig join algorithms *TwigStack* and *TwigStackList* respectively, based on the region labeling scheme. In order to answer a query twig pattern, the algorithms access the labels alone without traversing the original XML documents. *TwigStack* use a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths, it is I/O and CPU optimal algorithm that reads the entire input for twigs with only ancestor-descendant edges. *TwigStackList* overcomes the limitation of *TwigStack*, its main technique is to look-ahead read some elements in input data steams and cache limited number of them to lists in the main memory. *TwigStackList* is I/O and CPU optimal not only for queries with only ancestor-descendant edges, but when queries contain parent-child edges below non-branching nodes, the intermediate results can be guaranteed to be a subset of that in previous algorithms.

In this paper, motivated by the property the existing prime number labeling scheme [10] when determining the relationships of nodes, we proposed a new labeling scheme, called Pri-order, which can quickly determine the relationships between nodes and efficiently support the dynamic updating of XML document tree . To reduce the intermediate results size of query twig pattern further, a modified join algorithm: TLP is proposed based on the Pri-

Manuscript submitted July 5, 2006.
Manuscrip revised July 20, 2006.

order labeling scheme. The new algorithm has the same performance as *TwigStack* and *TwigStackList* when queries contain only ancestor-descendant edges, but more efficient than them for queries with the presence of parent-child edges below the branching nodes.

The rest of paper is organized as follows. We first discuss the related works about twig pattern in section 2. The novel algorithm is presented in section 3. We report the experimental results in section 4 and section5 we conclude this paper.

2. Related Works

2.1 Data Model and Query Twig Pattern

XML uses a tree structure model for representing data, an XML database is a forest of ordered and labeled trees, where nodes represent element, attributes and texts, and edges represent element-subelement, element-attribute and element-text relationships. Most existing query processing algorithms use a labeling scheme to present the information of position of a tree node. These labeling schemes can support efficiently evaluation of structural relationships. In our paper, we use Pri-order labeling scheme to label the node in the tree. We will explain the labeling scheme in section2.3. Figure 1 shows the tree representation of a XML document. We don't give labels (described in ()) of all nodes in the tree for the limitation of space.

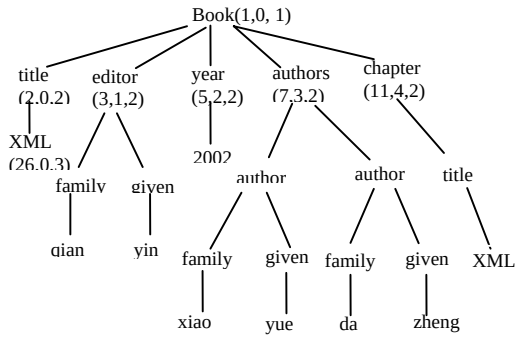

Fig. 1 Tree representation of a XML document

Most of XML query languages like XQuery [2] make use of twig pattern to match relevant portions of data in an XML database. Twig pattern nodes may be elements, attributes and texts. Twig pattern edges are either parent-children relationships (denoted by '/') or ancestor-descendant relationships (denoted by '//'). If a node has more than one child, then we call this node a branching node. Otherwise, when the node has only one child, it is a non-branching node. For example, the XQuery expression: Book [title = 'XML' and year= '2002'] can be represented as the twig pattern in Figure2 (a). Only parent-children edges are used in this case. Similarly, the XQuery expression: book [title = 'XML']//author [family = 'xiao' and given= 'yue'] can be represented as the twig pattern in Figure2 (b). Note that an ancestor-descendant edge is used between the book element and the author element, *title* is a non-branching node and *author* is a branching node in this twig pattern.

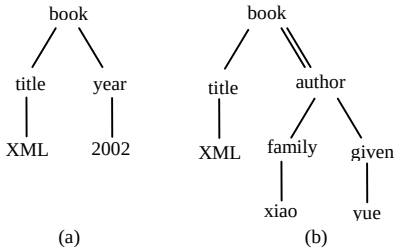

Fig. 2 Queries twig pattern

2.2 Twig pattern Match

Given a query twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from nodes in Q to elements in D, such that: (1) query node predicates are satisfied by the corresponding database elements, and (2)the parent-children and ancestor-descendant relationships between query nodes are satisfied by the corresponding database elements. The answer to query Q with m nodes can be represented as a list of m-ary tuples, where each tuple (q_1, q_2… q_m) consists of the database elements that identify a distinct match of Q in D.

For example, a query twig pattern in Figure2 (b), and the data tree in Figure 1, we need to find all the accurate occurrences of twig pattern of Figure2 (b) in the database corresponding to Figure 1.

2.3 Pri-order Labeling Scheme

XML tree structures must be preserved explicitly when XML documents was accessed. One method is to assign labels for the nodes in XML tree. We can capture the structural information of XML documents and perform the twig pattern matching based on the labels alone without traversing the original XML documents.

For designing a proper labeling scheme for XML documents, various methods have been proposed, they can be divided into two kinds: one is region-based [11, 12], the other is prefix-based [13, 14]. Most of joins algorithms are based on the region labeling scheme, which is not flexible

for processing the dynamic updating of XML document tree.

In this paper, we proposed a new labeling scheme for XML document based on the existing prime number labeling scheme, called *Pri-order* labeling scheme. It allocates a 3-tuple (*pri*, *ord*, *level*) for each node in XML document tree structure, to represent the position of node, it also can succinctly captures structural relationships between nodes in the XML database.

pri is unique integer gained with exploiting the prime number labeling scheme. The prime number labeling scheme exploits the unique property of prime numbers to label the nodes, in this labeling scheme each non-leaf node is given a unique prime number by width-first traversal of the XML tree, and the label of each node is the product of its parent node'label and its own allocated one, the label of node is divided only by its ancestor's label. Note that the first part of *Pri-order* utilizes the property of the prime number labeling scheme, is good for dynamic updates of document. When a new node is inserted, it is easy to assign a prime number that has not been assigned before as its own one for the new inserted node, no re-labeling is required in this part. In *Pri-order* labeling scheme the ancestor-descendant relationship between any two nodes is checked efficiently by this part of label of node, we only check whether the *Pri* of the ancestor node is divided by the *Pri* of the descendant node or not.

ord is a integer, denotes the sibling order of nodes with the same parent, the order of first child of node is zero. *ord* makes sure the order between nodes with the same parent node. In this scheme, only sibling order is re-labeled, when updating occurs in document, the *pri* and *level* are not affected. The range of re-labeling is small, only the following sibling nodes which have same parent node with inserted node need to be re-labeled.

level is the level where node locates, the level of the root node is one. By *pri* and *level* we can determine the parent-child relationships of nodes. For the relationships of any two nodes n_1 and n_2 in the document tree, n_1 is ancestor of n_2 if and only if n_2.pri mod n_1.pri = 0, for the parent-child relationship, we also check whether n_1.level =n_2.level-1.

3. Twig Join Algorithm

In this section, we present a new join algorithm for finding all matches of a query twig pattern against an XML document; it is a modification of *TwigStackList* based on the Pri-order labeling scheme. We introduce some notations and data structures which will be used by this join algorithm first.

3.1 Notation and data structures

A query twig pattern can be represented with a small tree. The node operations are defined as follows: function *isRoot*(n) examines a query node is a root or not, and *isLeaf*(n) examines a query node is a leaf node or not, *IsBranching(n)* examines whether a query node. The function *children*(n) gets all child nodes of n, and *PCRchildren*(n), *ADRchildren*(n) return child nodes which have the relationships of parent-child or ancestor-descendant with n in the query twig pattern, respectively. That is *PCRchildren*(n) ∪ *ADRchildren*(n) = *children*(n). The two operations *ancestor (e)* and *descendant (e)* over elements in the document return the ancestors and descendants of e, both including e respectively. In the rest of paper, "node" refers to a tree node in the twig pattern, while "element" refers to an element in the data set involved in a twig join.

There is a data stream T_n associated with each node n in the query twig. We use C_n to point to the current element in T_n. Function end (C_n) tests whether C_n is at the end of T_n. We can access the attribute values of C_n by $C_n.pri$, $C_n.ord$ and $C_n.level$. The cursor can be forwarded to next element in T_n with the procedure advance (T_n).Initially, C_n points to the first element of (T_n).

Our algorithm will make use of two types of data structure: list and stack. Given a query twig, we associate a list L_n and a stack S_n with each node n in the twig pattern.

The use of stack in our algorithm is similar to that in previous join algorithm, each data node in the stack consists of a pair :(positional representation of an element from T_n, pointer to an element in $S_{parent(n)}$). The operations over stack are: *empty*, *pop*, *push*, *top.pri*, the last operation returns the *pri* attribute of the top element in the stack. At every point during computation: (i) the nodes in stack S_n (from bottom to top) are guaranteed to lie on a root-leaf path in the XML database. (ii) The set of stacks contain a compact label of partial and total answers to the query twig pattern.

For each list L_n, we declare an integer variable p_n, as a cursor to point to an element in the list L_n, we use $L_{n.At(pn)}.pri$, $L_{n.At(pn)}.ord$ and $L_{n.At(pn)}.level$ to get the attributes of element in list L_n which p_n points to. At every point during computation: elements in each list L_n are strictly nested from the first to the end. The operations over list are *delete* (p_n) and *append (e)*, the first operation delete the element pointed by p_n in list L_n and the last operation appends element e at the end of L_n

3.2 Twig Join Algorithm: TJP

Algorithm TJP returns results to a query twig pattern, it operates in two phases. In the first phase it repeatedly calls the *getNext* algorithm with the query root as the

parameter to get the next node for processing. We output solutions to individual query root-to leaf paths in this phase. In the second phase, these solutions are merge-joined to compute the answer to the whole query twig pattern.

In section 3.2.1, we give the algorithm TJP, and section 3.2.2 presents the *getNext* algorithm, which was called in algorithm TJP.

3.2.1 Main Algorithm

In this section we will show algorithm TJP: a modified algorithm of *TwigSatckList* based on the Pri-ordering labeling scheme. It calls *getNext(n)* to get next processed node n repeatedly, when n is a branching node, we make sure the elements in list L_n must participate in the final solution, otherwise we will delete it from the L_n. the following describes the details of modified algorithm.

Algorithm TJP
Begin
1: while (! end (n)) do
2: g = getNext (root)
3: if (! isRoot (g)) then
4: clearStack($S_{parent(g)}$,g, getpri(g))
5: end if
6: if (isroot(g)∨!empty($S_{parent(g)}$))then
7: clearStack(S_n,g, getpri(g))
8: movetoStack(g, S_g, pointertotop($S_{parent((g))}$))
9: if (isLeaf(g)) then
10: showSolutions(S_g)
11: pop (S_g)
12: end if
13: else
14: proceed (g)
15: end if
16: end while
17: MergeSolutions ()
End

Function end ()
1: return $n_i \in$ descendant (n):isLeaf(n_i)∧ end(C_n);

Procedure movetoStack(n, S_n, p)
1: push (getElement(n), p) to stack S_n;
2: proceed (n);

Procedure clearStack (S_n,n, g.pri)
1: while (! empty (S_n) ∧ (g.pri mod toppri(S_n) !=0))do
2: pop (S_n);
3: end while

In algorithm TJP, line 2 calls *getNext* algorithm to identify the node to be processed. Line 4 and 7 remove partial answers from the stacks of parent (g) and g that can not contribute to the final answer. If n is not a leaf node, we will push element e_g into S_g, otherwise output all path solutions involving element e_g(line10), the individual solutions should be output in root-to-leaf order so that they can be easily merged together to get final twig query results(line17).

3.2.2 getNext Algorithm

Algorithm getNext (n)
Begin
1: if (isleaf (n)) return n;
2: for each node n_i in children (n) do
3: r_i=getNext (n_i);
4: if (ri! =n_i) return r_i;
5: end for
6: n_{max}= maxarg $_{ni \in children(n)}$ getpri(n_i);
7: n_{min}=minarg $_{ni \in children(n)}$ get pri(n_i);
8: while (getpri (n_{max}) mod getpri (n)! =0) proceed (n);
9: if (getpri (n) mod getpri(n_{min}) =0) return n_{min};
10: moveStreamtoList(n,n_{max})
11: if (isBranching(n))
12: for each e_i in Ln ∈MB (n_{min},n)
13: if (e_i! ∈ancestor (e_{max})) delete (Ln,e_i);
14: end for
15: end if
16: for each node n_i in children (n) do
17: if (there is an element e_i in list L_n such that e_i is the parent of getElement(n_i))then
18: if (n_i is the only child of n) then
19: move the cursor p_n of list L_n to point to e_i;
20: end if
21: else return n_i;
23: end if
24: end for
25: return n;
End

Procedure getElement(n)
1: if (! empty (L_n) then
2: return Ln.elementAt(p_n);
3: else return C_n;

Procedure getPri(n)
1: return the pri attribute of getElement(n);
Procedure moveStreamtoList(n,m)

1: while (getpri(m) mod C_n.pri =o) do
2: L_n.append(C_n);
3: advance (T_n);
4: end while

Procedure proceed (n)
1: if (empty (L_n) then
2: advance (T_n);
3: else

4: Ln.delete(T_n);
5: p_n=0{move p_n to point to the beginning of L_n}
6: end if

Function MB (n,b)
1: if (isBranching(n)) then
2: let e be the maximal element in list L_n
3: else
4: let e =current (T_n)
5: end if
6: return a set of element a that is an ancestor of e such that a can match node b in the path solution of e to path pattern p_n

Procedure deleteList (L_n, e)
Delete any element a in the list Ln such that a! ∈ancestors (e) and a! ∈descendants (e)

The *getNext*(n) is a procedure called by the join algorithm TJP, it returns a node n' (possibly n' = n) with three properties: assume that element $e_{n'}$= getElement(n') then(i)$e_{n'}$ has a descendant in each of stream T_{ni} for $n_i \in$ children(n'); and (ii) if n' is not a branching node in the query, element $e_{n'}$ has a child e_{ni} in T_{ni}, where $n_i \in$ *PCRchildren*(n')(if any); and (iii) if n' is a branching node, it must contribute to the final answer of query, there is an element e_{ni} in each T_{ni} such that there exists an element e_i(with tag n)in the path from $e_{n'}$ to e_{nmax} that is the parent of e_{ni}, where $n_i \in$*PCRchildren*(n')(if any) and e_{nmax} has the maximal pri attribute for all *children*(n')

At line2-5, the algorithm *getNext* repeatedly calls itself for each $n_i \in$ children (n). If the returned node r_i is not equal to n_i, the algorithm immediately return r_i. Otherwise, it will try to get a child of n which satisfies the above three properties. Line6 and line7 get elements which have max and min pri attribute for the current head elements in lists or streams, respectively. Line 8 skips the elements that do not contribute to the final results. If no common ancestor for all C_{ni} is found, Line9 returns the child node with the smallest pri value. At line 10, we look-ahead some elements in the stream T_n and cache elements that are ancestors of C_{nmax} into the list L_n.

Line11-15 are important steps, for the branching node element e_i in list L_n, if it does not participate in the solution of the future elements in other streams, we will delete it from the list L_n. Note that these steps are key difference between *TwigStackLis*t and TJP. By these steps we can make sure the elements in L_n must contribute to the final solutions, the previous one will return the branching nodes that may result in many "useless" intermediate results. Algorithm TJP evaluates all branching nodes; they will be deleted from cache list L_n when they don't participate in the final answer.

Example1. Find a query twig pattern on a document in figure 3.TwigStackList will cache elements a_1 and a_2 in list L_a, c_1 and c_2 in list L_c, then moves a_1, c_1 and c_2 into stack S_a and S_c, respectively. Different from *TwigStackList*, the TJP algorithm will delete c_2 from list L_c, only pushes c_1 into list L_c, because c_2 does not contribute to the final answer of query, so it does not output the solution<c_2, e_1, g_1>. At last our algorithm merges only individual path solutions<a_1, b_1> and <a_2, c_1, d_1, f_1, e_1, g_1> to get the final query result.

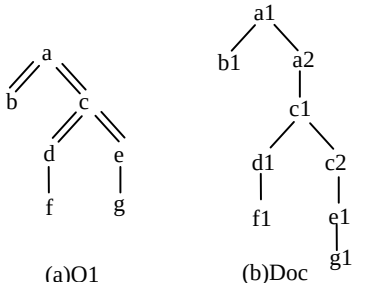

Fig. 3 Example twig query and documents

3.2.3 Analysis of Algorithm

In this section, we discuss the correctness and the complexity of TJP, and then we analyze its complexity. Finally, we compare TJP with *TwigStackList* in terms of the size of intermediate results.

TJP is similar to *TwigStackList*, for any node n in the twig query we have *getNex*t (n) =n'. Then (1) n' has the child and descendant extension. (2) either (i) n =n' or (ii) parent (n) does not have the child and descendant extension because of n' or a descendant of n'.

Lemma1. In procedure *deleteList* of Algorithm *getNext*, any element e that is deleted from list L_n does not participate in any solution.

Lemma 1 shows that any element deleted from list dose not participate in individual root-leaf solutions, so it doesn't contribute to the final result, the deletion is safe. Any element e that matches a branching node, if e participates in any final answer, then e occurs in the corresponding list.

Lemma2. At our algorithm TJP, elements in stack S_n are strictly nested; each element is a descendant of the element below it.

We push new element into the stack only in Procedure *movetoStack*, there are two cases for relationship between the new element e_{new} to be pushed into stack and the existing top element e_{top} in stack.

Case 1: There is not the ancestor-descendant relationship between two nodes; the element e_{top} will be

popped in procedure *clearStack*. So the case is impossible.

Case 2: They are the ancestor-descendant relationship; in this case, e_{new} will be pushed into the stack safely.

Lemma3. In TJP, any element which is popped from the stack S_n does not contribute to any new solution any more.

For the first call of *clearSatck* in line 4 of main algorithm, suppose on the contrary, there is a new solution involving the popped element e_{pop}, we can note line 1 of *clearStack*, if g.pri mod e_{pop}.pri! = 0, the element e_{pop} will be popped from the stack. And e_{pop} can not involved in any element in the path from the root to $e_{parent(n)}$ and after $e_{parent(n)}$, it is a contradiction.

For the second call of *clearStack* in line 7, according to the containment property, the elements popped form the stack are descendant of e_n, they don't participate in any new solutions any more. There no element is a child of e_{pop} in the rest of elements in $T_{children(n)}$, so e_{pop} does not participate in any new solutions.

These lemmas are important to determine the correctness of the following theorem.

Theorem1. Given a twig query Q and an XML database D, Algorithm correctly returns all the answers for Q on D.

Proof (sketch): Using the lemmas, we know that when getNext returns a query node n, if the stack of n'parent is empty, the head element of L_n does not contribute to any final solutions. Any element in the ancestors of n that use e_n in the descendant and child extension is returned by getNext before e_n. For branching nodes, we make sure they must contribute to the finial solutions before they was push to stack, finally, when n was returned by getNext is a leaf node, we output all solutions that use e_n.

While the correctness holds for query twig patterns with both ancestor-descendant and parent-child relationships in any edges, *TwigStackList* can be proved optimality only for which parent-child relationships appear only in edges below non-branching nodes, thus algorithm TJP is optimal for parent-child relationships below branching node, according to lemma 2, we are guaranteed that branching node e_n is pushed into stack, only when e_n has really participated in the finial solutions.

Theorem2. Consider a query twig pattern Q with m nodes and an XML database Q. the worst case I/O complexity of algorithm is linear in the sum of the size of input and output lists. The worst case space complexity is proportional to m times of the maximal length of a root-leaf path in D.

For the limitation the space, we do not give the proof the theorem 2. It is important to note that, we delete the branching node element which does not participate in the final solutions from the list. The experiment results were shown in section 4, our join algorithm output less intermediate solutions than *TwigStackList*. The reason is our algorithm deleted the branching node elements which don't contribute to the final answers, thus, it pushed fewer elements into stack and thereby less intermediate results were outputted.

4. Experiment Evaluations

We present experimental results on the performance of the twig pattern matching algorithms in this section with real datasets. We evaluated the performance of these algorithms using the following datasets, experiment checked the size of intermediate results compare our algorithm with *TwigStack* and *TwigStackList*.

4.1 Experimental Setting

We implemented all algorithms in JAVA. All our experiments were performed on Pentium 4 2.4 G processor; with 512MB RAM running on windows XP system. We used the datasets DBLP and TreeBank for our experiment; the two datasets have different properties: DBLP is a shallow and wide document, but TreeBank has very deep recursive structure. We summarize the properties of two datasets in table1.

Table 1: XML Datasets

	DBLP	TreeBank
Data size(MB)	130	22.8
Nodes(million)	3.3	2.4
Max/Avg depth	6/2.9	36/7.8

4.2 Twig Queries

Now we give the experimental results of twig queries, compared TJP with the *TwigStackList*. We tested several XML query on DBLP and TreeBank data in table 2. These queries have different twig structures and combinations of parent-child and ancestor-descendant relationships. In particular, query Q1 contains only ancestor-descendant relationships, while Q2 contains only parent-children relationships. Q3contains only ancestor-descendant relationships between the branching node and its children, while Q4 contains a branching node with both parent-child and ancestor-descendant relationships.

Table 2: X Twig queries on DBLP and TreeBank

	Dataset	Twig queries
Q1	DBLP	//article[//sup]//title//sub
Q2	TreeBank	/S/VP/PP[IN]/NP/VBN
Q3	TreeBank	/S[//VP/IN]//NP
Q4	TreeBank	//VP[DT]//PRP-DOLLAR

We analyze the query performance of algorithms *TwigSatck*, *TwigStackList* and *TJP*, all the join algorithms need to scan the all elements for nodes; they have the same

cost of disk access. Thus, for the size of intermediate results, we mainly examined the results of queries with parent-child relationships. Table 3 shows the number of intermediate solutions and the final answers of queries Q2, Q3, Q4. From table 3, we can see, all join algorithms are all sub-optimal. The number of intermediate path solutions of *Twigstack* is largest, over 95% useless intermediate solutions generated by it. The other two algorithms have almost same performance, the intermediate solutions generate by them are slightly large than the number of useful solutions. But compared with the *TwigStack* and *TwigStackList*, TJP is more efficient, especially query with parent-child relationships below the branching nodes, like query Q2.

Table 3: Number of intermediate solutions

Algorithm / Query	Twig stack	TwigStack List	TJP	Useful
Q2	2237	388	337	302
Q3	702391	22565	22565	22565
Q4	10663	9	9	5

On the other hand, we also can check the query execution time of join algorithms based on different labeling scheme. Figure 4 shows the execution time of TJP and *TwigStackList* for queries Q2, Q3, Q4, the two algorithms are based on the region labeling and the Pri-order labeling scheme respectively.

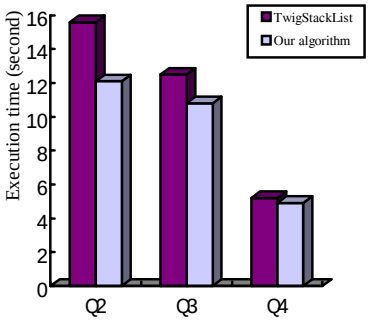

Fig. 4 Queries execution time of two join algorithm

From the figure 4, we can see the algorithm TJP adopts the Pri-order labeling scheme, the relationships between nodes can quickly be determined; it can reduce the time for reading the input. For the number of branching nodes is reduced in stack, it also can save up the execution time.

Compared all results of experiment we note that TJP outperforms *TwigStack* and *TwigStackList* under the dataset. The improvement is due to the facts that for query with branching nodes, they must contribute to the final answer before the branching nodes were moved to the list. With this technique, the size of intermediate results is reduced; and use the advantage of the labeling scheme for determining the relationships among nodes, the query execution time is saved up.

5. Conclusions

XML twig pattern matching is a key issue for XML query processing. In this paper, we propose an efficient algorithm for query XML twig pattern, called TJP. It is a modification of *TwigStackList* using Pri-order labeling scheme. This labeling scheme can determined the relationship of any nodes quickly without traveling the original document; it is insert-friendly in the context of dynamic update of XML trees. The most important technique of TJP is when processing query with parent-child relationships below branching node, we guaranteed that the branching node elements in list must contribute to the finial answers, or else we delete it from list. The number of intermediate path solutions for query twig pattern is much smaller than previous algorithms. Compared with *TwigStackList* and *TwigStack*, the experimental results showed that our method TJP is more efficient, especially for queries with parent-child edges below the branching node.

Acknowledgment

As authors, we would like to express our cordial thanks to reviewers for their valuable advice.

References

[1]A.Bergund, S.Boag, et al. XML Path Language (XPath) 2.0, W3C Working Draft 22 August 2003.
[2]S.Boag, D,Chamberling et al.XPaht 1.0: An XML Query W3C , Working Draft 22 August 2003.
[3]N.Bruno,D.Srrivastave,D.Koudas. Holistic Twig Joins: optimal XML Pattern Matching. In proceedings of ACM SIGMOD, 2002, P310-321.
[4]S.Al-Khalifa, H.V.Jagadish, N.Koudas et al. Structural joins: A Primitive for efficient XML query pattern matching, In Proceedings of ICDE conference, 2002, P141-152.
[5]J.Mchugh, J.widom. Query Optimization for XML. In Proceedings of VLDB, 1999, P315-326.
[6]T.chen, J. Lu, T. Ling. On Boosting Holism in XML Pattern Matching using Structural Indexing Techniques. In Proceedings of ACM SIGMOD, 2005, P455-466.
[7]H.Jiang,W.Wang, H.Lu. Holistic twig joins on indexed XML documents. In proceedings of VLDB, 2003, P273-284.
[8]C.Zhang, J.Naughton, D.Dewitt et al.On Supporting Containment Queries in Relational Databases Management System. In Proceedings of ACM SIGMOD, 2001, P425-436
[9]J. Lu, T.Chen, T.Ling. Efficient Proceeding of XML Twig Patterns with Parent-Child Edges: a Look-ahead Approach. In CNKI, 2004, P533-542.

[10]X.Wu, M.Led, W.Hsu. A Prime Number Labeling Schemes for Dynamic Ordered XML Trees Proceeding of the 20th International Conference on Data Engineering ICDE ,2004: 66-78.
[11]SihemAmer-Yahia,MaryFernandex,Davesh Srivastava,Phase Matching in XML, proceeding of the 29th VLDB Conference Berlin Germany,2003; 177-188.
[12] Dietz PF. Maintaining order in a linked list. Proc of the Annual ACM Symposium on Theory of Computing SanFrancisco, Cali-fomia, May l982:122-127.
[13] E.Cohen,H.KapLan and T.Milo, Labeling Dynamic XML Tree ,In PODS,2002:271 – 281.
[14]H.Kaplan, T.Milo and R.Shabo, A comparison of Labeling Schemes for Ancestor Queries in SODA,2002, 954-963.

Ren Jiadong received the B.E and M.E degrees, from Northeast Heavy Machine College in 1989 and 1994; respectively.He received the Dr. Comp. degree from Harbin Institute of Technology Univ. in 1999. After working as a teacher (from 1989), an assistant professor (from 1999) in the Yanshan Univ., and he has been a professor at Yanshan Univ. since 2005. His research interest includes Space-time Database, XML Data Model and Data Mining, and their applications. He is a member of IEEE SMC Society China and director of Electrical Education Consortium of Hebei Province of the China.

Yue Liwen received the B.E from Jilin Normal Univ. in 2000. She is a postgraduate of YanShan Univ. from 2000. Her main research interest is the query and storage of XML data