WUJNS
WUHAN UNIVERSITY JOURNAL OF NATURAL SCIENCES

Vol. 8 No.2B 2007 603-609

Article ID:

TwigStack+: Holistic Twig Join Pruning Using Extended Solution Extension

□ **ZHOU Junfeng[1,2], XIE Min[1], MENG Xiaofeng[1]**

[1]School of Information, Renmin University of China, 100872, Beijing

[2]Department of Computer Science and Technology, Yanshan University, Qinhuangdao, 066004, China

Abstract: XML has been used extensively in many applications as a de facto standard for information representation and exchange over the internet. Huge volumes of data are organized or exported in tree-structured form and the desired information can be got by traversing the whole tree structure using a twig pattern query. A new definition, Extended Solution Extension, is proposed in this paper to check the usefulness of an element from both forward and backward directions. Then a novel Extended Solution Extension based algorithm, TwigStack+, is also proposed to reduce the query processing cost, simply because it can check whether other elements can be processed together with current one. Compared with existing methods, query evaluation cost can be largely reduced. The experimental results on various datasets indicate that the proposed algorithm performs significantly better than the existing ones.

Keywords: XML; Database; holistic join

CLC Number: TP 391

Received date: 2007-04-25

Foundation item: supported by the grants from the Natural Science Foundation of China (60573091, 60273018)

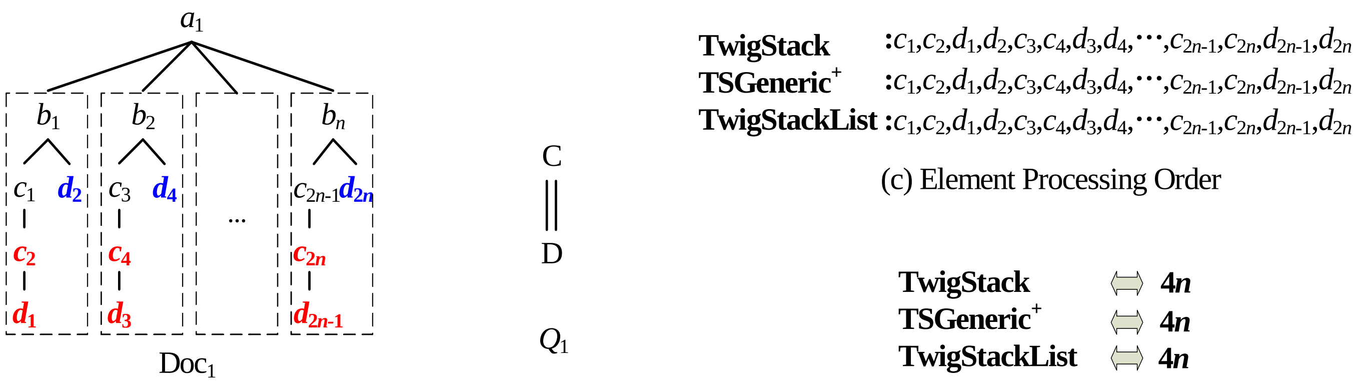
Biography: ZHOU Junfengg(1977-), male, PhD student, research direction: XML Database. Email: zhoujf@ysu.edu.cn

0 Introduction

As a de facto standard for information representation and exchange over the internet, XML has been used extensively in many applications. An XML document contains hierarchically nested elements, thus it can be naturally modeled as a tree, where nodes represent elements while edges represent direct nesting relationships between elements. Query capabilities are provided through twig queries, which are the core components of standard XML query languages, e.g. XPath[1] and XQuery[2]. A twig query can be modeled as a node labeled tree, where nodes are labeled with different tags, and edges represent either the parent-child (P-C) or ancestor-descendent (A-D) relationships. Existing query processing methods[3~8] can process a twig query very efficiently; however, they still suffer from large number of redundant function calls. We propose a new holistic twig join algorithm, which can greatly improve query processing performance, to solve this problem.

1 Related Work and Analysis

Recently, Holistic Twig Join methods[3~8] have been proposed to process a twig query efficiently by avoiding large number of intermediate results using a chain of linked stacks to compactly represent partial results of individual query paths. All these methods are based on tag index which was organized using XB-tree, XR-Tree or B+-Tree and the whole process of twig pattern matching consists of two stages: (i) Find the partial results of each individual query paths and (ii) Merge all the partial results to form the final results. To find partial



(*a*) An XML Document (*b*) A Twig Query (c) Element Processing Order (*d*) Calling times of getNext(root)

Figure 1. XML document, query example and the processing

results of each individual query path in the first stage, getNext(f), the most important procedure, is called repeatedly in the first stage to get a query node q which has Solution Extension and then, we can get an element C_q which has the smallest document order among all the unprocessed elements, all elements before C_q without Solution Extension[1] is skipped directly. When there are only ancestor-descendant edges in the given twig pattern, TwigStack[3] can guarantees the optimality of I/O and CPU time, iTwigStack[4] can improve the query processing performance by using structural index to filter out useless data elements. TSGeneric+[5] made improvements on TwigStack by using XR-Tree to skip some useless elements which have Solution Extensions but cannot participate in any path solution. TwigStackList[6] handles the sub-optimal problem by attaching an element list to each query node to cache some elements, TJFast[7] improved the query processing performance by scanning elements of leaf nodes in the query to reduce the I/O cost, the work proposed in [8] can avoid the expensive cost of merging operation in the second phase by using a hierarchical stack to buffer temporal elements.

Although the existing methods[3,5,6] can guarantee the optimality of CPU time and I/O when only AD edges involved in the twig pattern, they all suffer from large number of redundant calls of getNext(root). For example, the XML document Doc_1 is shown in figure 1(a), when we evaluating the query Q_1 in figure 1(b), although all the elements returned by getNext(root) in figure 1(c) have *Solution Extension*, the elements, represented by $d_{2i}(1 \le i \le n)$, are useless elements since they cannot participate in any single path solution, as a result, all these calls of getNext(root) in *TwigStack* are useless. As stated before, TSGeneric+ can skip some useless elements in some cases, but for Doc_1 and Q_1 in figure 1(a) and (b), all these useless elements, represented by d_{2i}, cannot be skipped. For TwigStackList, the elements returned by getNext(root) are same as that in TwigStack and TSGeneric+. For the $4n$ elements returned by getNext(root), the number of calling getNext(root) in each method is $4n$, these redundant operations caused by getNext(root) should have been avoided. Moreover, after c_{2i-1} is processed, c_{2i} can be processed immediately without being returned by another call of getNext(root), further, d_{2i-1} can also be processed without calling getNext(root). As a result, we can reduce the calling time of getNext(root) to n which will positively affect query performance.

The reason for the phenomena described above lies in that the operation of getNext consists with the definition of Solution Extension, which can only guarantee the sub query rooted at q composed entirely of the cursor elements of the query nodes in the sub query form a partial results, i.e. if q is not the root node, elements correspond to q with Solution Extension may not participate in any final results.

2 Extended Solution Extension

The previous methods[3,5,6] use function getNext(n_{root}) to return an element which has Solution Extension for processing, but the returned element may not contribute to any final result since it may not have a proper ancestor node to form a matched result. To make getNext more efficient, we propose a new definition for Solution Extension to guide the operation of getNext.

Let q be a query node and C_q be an element

1 We the simplicity of expressiveness, we say an element has Solution Extension if the corresponding query node has Solution Extension

corresponding to q, parent(q) be the parent query node of q. We define Extended Solution Extension as follows:

Definition 1. (Extended Solution Extension): If q is the root query node, C_q has an Extended Solution Extension iff C_q has a Solution Extension, Otherwise, C_q has an Extended Solution Extension iff: (i) C_q has a Solution Extension and (ii) There is an element $C_{\text{parent}(q)}$ which has Extended Solution Extension and satisfies the structural relationship with C_q which is specified by the query pattern between parent(q) and q.

Lemma 1. If an element node C_q has an Extended Solution Extension, it must participate in the final results when only A-D edges are involved in the query pattern.

3 *TwigStack*$^+$: An Efficient Holistic Twig Join Method

The new algorithm TwigStack$^+$ presented in Algorithm 1 uses Extended Solution Extension to guide the process of getNext(root), which can guarantee that any returned element has an Extended Solution Extension, as a result, the returned element from getNext must participate in the final results when only A-D edge is considered, which in turn can avoid many redundant call of getNext(root). Moreover, we use Extended Solution Extension to optimize the push operation in TwigStack$^+$ to further reduce large number of redundant call of getNext(root).

Assuming each query node q is associated with a stack S_q, a cursor C_q and a data stream T_q. C_q can pointed to some element in T_q, we use a triple (start, end, level) to denote an element in T_q and we can use C_q.start, C_q.end and C_q.level to get the element's attribute value. Before executing the algorithm, all cursors point to the first elements in each data stream, we can use Advance(C_q) to make C_q point to next element. The self-explaining functions isRoot(q) and isLeaf(q) examine whether q is a root node or a leaf node. The function children(q) gets all child nodes of q.

As shown in algorithm 3, getNext(root) returns an element node which has an Extended Solution Extension. To achieve this goal, we first check each descendant node n_i of n whether it has Solution Extension in line 1-4. If any n_i' does not equal to n_i, we can guarantee that $C_{ni'}$ has an Extended Solution Extension, so we can safely send it to its outer procedure. Otherwise, all head element nodes of n's descendant query nodes have Solution Extension, there are two kinds of scenarios: (i) There is an element node in T_n which has a Solution Extension(line 9), we can return it to the outer procedure, because if n is the root query node it must have an Extended Solution Extension, otherwise, it is only an intermediate query node which has a Solution Extension and we can safely send it to the parent query node; (ii) we may need to either return the corresponding head element node in $T_{ni'}$ which has an Extended Solution Extension (line 10) or skip the element which does not participate in the final results(line 13-16).

When an element is returned to TwigStack$^+$ in line 2 of Algorithm 1, the main difference between TwigStack$^+$ and TwigStack is that in TwigStack, we only push the current element C_q into the stack, but in TwigStack$^+$ presented in Algorithm 2, we iteratively fetch elements, which also have Extended Solution Extensions and descendant-ancestor relationship with C_q, from T_q and push them into S_q (line 3-8).

```
Algorithm 1 TwigStack+(n_root)
 1:while not end(n_root)
 2:     q= getNext(n_root)
 3:     if (not isRoot(q))
 4:          cleanStack(S_parent(q),C_q)
 5:     cleanStack(S_q, C_q)
 6:     moveStreamToStack(q)
 7:mergeAllPathSolutions()
Algorithm 2   moveStreamToStack(q)
 1:while(C_q.end < C_parent(q).start)
 2:   if(C_n.isAncestorOfAllChildEleOf(q))
 3:     push(C_q,S_q,top(S_parent(q)))
 4:     if(isLeaf(q)) showSolutionsWithBlocking(C_q)
 5:     Advance(C_q)
 6:   elseif(C_n.end<C_n.MinChildEle().start)   Advance(C_q)
 7:   else break;
 8:for q_i∈children(q)
 9:   moveStreamToStack(q_i)
Algorithm 3 getNext(n)
 1:if(isLeaf(n))     return n
 2:for n_i∈children(n)
 3:     n_i' =getNext(n_i)
 4:     if(n_i <>  n_i' )     return  n_i'
 5:while(TRUE)
 6:     n_min = minarg_ni' { C_ni' .start}
 7:     n_max = maxarg_ni' { C_ni' .start}
 8:     C_n.fwdToAncestorOf(C_nmax)
 9:     if(C_n.start<C_nmin.start) return n
10:     cleanStack(S_n, C_nmin)
11:     if(not isEmpty(S_n))
12:          return n_min
13:     while(not C_n.isAncestorOf(C_nmin))
14:          if(C_n.start<C_nmin.start)
15:               C_n.fwdToAncestorOf(C_nmin)
16:          else C_nmin.fwdBeyond(C_n)
17:     getNext(n_min)
```

Example 1. Consider Q_1 and Doc_1 in figure 1 again, the first call of getNext(C) will return c_1 which has Extended Solution Extension. Since c_1 corresponds to root query node C in Q_1, it can be pushed into S_C and then C_C is moved to c_2, since c_2 and c_1 has the same Extended Solution Extension with d_1, in moveStreamToStack, c_2 is also pushed into S_C. Further, d_1 is also pushed into S_D, the current elements corresponding to C_C and C_D is c_3 and d_2. Although d_2 has Solution Extension, D will not be returned in the next call of getNext(C) since d_2 have no Extended Solution Extension, we directly forward C_D to d_3. Then the second call of getNext(C) will return element c_3 which has an Extended Solution Extension. As a result, we will push three elements into stack, and the calling times of getNext(root) in TwigStack$^+$ is n but not $4n$.

Theorem 1. Given a query twig pattern T and an XML database D, algorithm TwigStack$^+$ correctly returns all answers for T on D.

The intuition is obvious, if the current element cannot participate in any path solution, it will be directly discarded in getNext. Further, any element C_q which satisfies the axis requirement with top($S_{\text{parent}(q)}$) will be pushed into stack according to algorithm 1. At last, if an element can participate in any useful path solution, the path solution will be outputted according to algorithm 1. While correctness holds for twig patterns with both A-D and P-C edges, we have the following result.

Theorem 2. Consider a twig pattern T with n nodes, and an XML database D. Algorithm TwigStack$^+$ has worst-case CPU time complexity linear with the sum of sizes of the n input lists and the output lists. Further, the worst-case space complexity of TwigStack$^+$ is proportional to the minimum of (i) the sum of sizes of the n input lists, and (ii) n times the maximum length of a root-to-leaf path in D.

4 Experiments

4.1 Experimental Setup

All of our experiments are implemented on a PC with Pentium4 2.8 GHz CPU, 512 MB memory, 160 GB IDE hard disk and Windows XP professional as the operation system. In addition to TwigStack$^+$, we also implemented TwigStack, TwigStackList and TSGeneric$^+$ for comparison, further, we implemented TwigStack$^+$B, a B$^+$-tree based TwigStack$^+$ which can skip useless elements in element stream like TSGeneric$^+$. All these algorithms are implemented using Microsoft Visual C++ 6.0.

We used XMark[9] and TreeBank[10] for our experiments. XMark is a well-known synthetic XML data set and it features a moderately complicated and fairly irregular schema, with several deeply recursive tags. TreeBank has a deep recursive structure, which makes it ideal for experiments of twig pattern matching algorithms. The queries used in our experiment are listed in Table 1. The size of XMark dataset used in our experiment is 113MB and TreeBank dataset is 84M.

4.2 Performance Comparison and Analysis

(1) Performance Measures. We consider the following performance metrics to compare the performance of twig pattern matching algorithms: (1) Call times of getNext(root). This metric reflects the ability of selected algorithms to reduce the CPU cost for different kinds of query patterns(though the CPU cost is not in proportion to the call times of getNext(root)). (2) Running time. This metric is the most important one for evaluating the integrated performance of selected algorithms. The experimental results are shown in table 2 and figure 2(a~b).

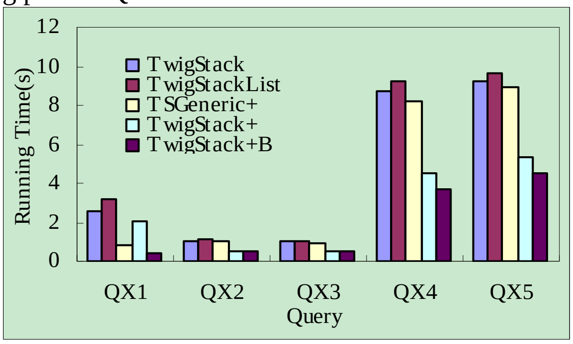
Table 1. Queries used in our experiment

Query	Data Set	XPath Expression
QX1	XMark	/site/closed_auctions/closed_auction[/annotation/description/parlist/listitem/text/keyword/bold]/price
QX2	XMark	/site/people/person[/profile[/education]/age]/phone
QX3	XMark	/site/people/person[//age]//education
QX4	XMark	//listitem[//bold]/text//emph
QX5	XMark	//listitem[//bold]/text[//emph]/keyword
QT1	TreeBank	//VP[/DT]//PRP_DOLLAR_
QT2	TreeBank	//S[/JJ]/NP
QT3	TreeBank	//S/VP/PP[/NP/VBN]/IN
QT4	TreeBank	//S//NP[/PP][//VP]//JJ
QT5	TreeBank	//S/VP[//NN]/VBD

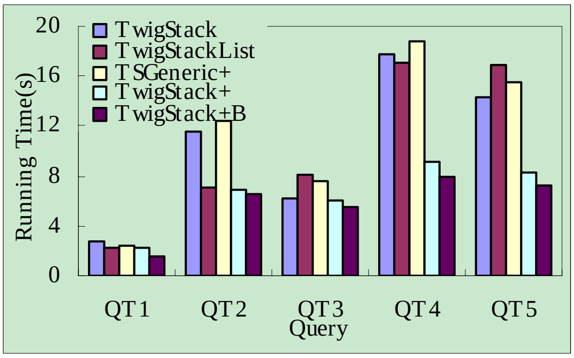
Table 2. The comparison of call times of getNext(root)

查询编号 / 算法	QX1	QX2	QX3	QX4	QX5	QT1	QT2	QT3	QT4	QT5
TwigStack	98500	30532	16224	203384	249287	137318	612632	207998	416541	299598
TwigStackList	95717	30532	16224	203037	245346	127583	521447	171609	401302	287674
TSGeneric+	9187	13569	11984	103436	100208	32858	519203	116876	124630	176970
TwigStack+	1054	1630	3275	42115	39825	15175	331791	15970	78993	77734
TwigStack+B	1054	1630	3275	42115	39825	15175	331791	15970	78993	77734

(2) Scalability. We also test the five algorithms on data sets of different sizes. We present in table 3 and figure 3 the performance results tested on XMark benchmark size of 12MB, 58MB, 113MB and 174MB and 232MB on the twig pattern QX5.



(a) XMark



(b) TreeBank

Figure 2. The comparison running time

Table 3. The comparison of call times of getNext(root) over different datasets of different size using QX5

文档大小 / 算法	12M	57M	113M	174M	232M
TwigStack	25011	124296	249287	375293	500055
TwigStackList	24579	122211	245346	369253	492061
TSGeneric+	10310	49942	100208	151231	201680
TwigStack+	4008	19795	39825	60303	80513
TwigStack+B	4008	19795	39825	60303	80513

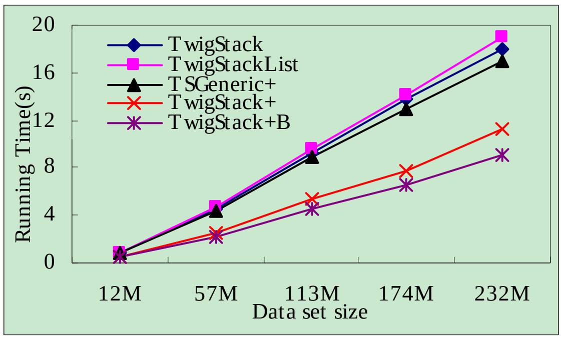


Figure 3. The comparison of five algorithms over different data sets of different size using QX5

(3) Performance Analysis. Compared with existing methods, as shown in table 2, it is obvious that by returning query nodes with Extended Solution Extension, the call times of getNext(root) in TwigStack+ and TwigStack+-B is reduced significantly. Although TSGeneric+ can skip on the input data streams, the call times of getNext(root) is still much larger than that in TwigStack+. For all queries, the call times of getNext(root) in previous methods are 1.5 to 100 times more than that in our methods.

When considering running time of various algorithms (figure 2(a) and (b)), we can see that TwigStack+ again achieves faster running time than TwigStack and TwigStackList. The reasons lies in two aspect: (i) getNext guided by Extended Solution Extension can save partial CPU cost, (ii) our push operation can further reduce the CPU cost. Although TSGeneric+ can improve query performance significantly for some queries by skipping on the input streams, e.g. QX1, our B+-tree based method, i.e. TwigStack+-B, can do much better than TSGeneric+. From figure 2(b) we can see that TwigStackList can get better performance for queries over documents of very complex structure, e.g. QT1, QT2 and QT4, this is because it can prevent some elements which will not participate in any final results from being pushed into stack, thus the CPU cost can be saved. However, by combining the savings from two aspects: getNext guided by Extended Solution Extension and push operation, TwigStack+ and TwigStack+-B can achieve faster running time than TwigStack, TwigStackList and TSGeneric+ (figure 2(a) and (b)).

5 CONCLUSIONS

Twig pattern matching is a core operation in XML query processing. Holistic twig join algorithms have been proposed recently to tackle the problem, but they all suffer from the great burden of redundant function call. We propose a novel holistic twig join algorithm TwigStack+ guided by Extended Solution Extension to avoid the redundant function call. Experimental results on various datasets indicate that the proposed algorithm performs significantly better than the existing ones.

6 REFERENCES

[1] Anders B, Scott B, Don C, et al. XML Path Language (XPath) 2.0 [R]. W3C, http://www.w3.org/TR/xpath20/, 2007.

[2] Scott B, Don C, Mary F. F, et al. XQuery 1.0: An XML Query Language[R]. W3C, http://www.w3.org/TR/xquery/, 2007

[3] Nicolas B, Nick K, Divesh S. Holistic Twig Joins: Optimal XML Pattern Matching [C]. // *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. Wisconsin, June 3-6, 2002.

[4] Ting C, Jiaheng L, Tok W. L. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques [C]. // *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. Maryland, June 14-16, 2005.

[5] Haifeng J, Wei W, Hongjun L, et al. Holistic Twig Joins on Indexed XML Documents [C]. // *Proceedings of 29th International Conference on Very Large Data Bases*. Berlin, September 9-12, 2003.

[6] Jiaheng L, Ting C, Tok W. L. Efficient Processing of XML Twig Patterns with Parent. Child Edges: A Look-ahead Approach[C]. // *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management*. Washington DC, November 8-13, 2004.

[7] Jiaheng L, Tok W. L., Chee Y. C., et al. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching [C]. // *Proceedings of 31st International Conference on Very Large Data Bases*. Norway, August 30 - September 2, 2005.

[8] Songting C, Huagang L, Junichi T, et al. Twig2Stack: Bottom-up Processing of Generalized –Tree-Pattern Queries over XML Documents [C]. // *Proceedings of 32nd International Conference on Very Large Data Bases*. Seoul, September 12-15, 2006.

[9] Ralph B. XMark[EB/OL]. http://monetdb.cwi.nl/xml/. March 2003

[10] Ann T. TreeBank.xml[EB/OL]. http://www.cs.washington.edu/research/xmldatasets/ . February 1999