2012 IACSIT Hong Kong Conferences
IPCSIT vol. 30 (2012) © (2012) IACSIT Press, Singapore

Xml Query with Parent-Child Edges Optimization Algorithm

Wang Jing-Bin[+]

College of Mathematics and Computer Science Fuzhou University Fuzhou, China

Abstract. XML has already been the de facto standard for data exchange because of its flexibility and expressiveness in Internet. Holistic twig join algorithm is an important method to query over XML document in XML Database Management System. In this paper, we propose a new algorithm PCTwigStackList, which is quite efficient to query with twig which include parent-child edges. In our algorithm, we use a data structure *List* to cache a group XML elements, and we can distinguish which element is impossible to participate in twig joins. In order to reduce the useless path solution of twig, our algorithm postpones outputting the individual path solution and judges these elements in *Stack* whether they will be to participate in final join. Thus we can reduce the CPU cost of Twig Joins algorithm and improve the efficiency of query with parent-child edges.

Keywords: XML, twig, holistic join, XML query.

1. Introduction

XML has already been the de facto standard for data exchange because of its flexibility and expressiveness in Internet. In the area of XML Database Management System, XPath and XQuery have been widely used in XML Query Language. XPath and XQuery are always modeled as a Twig. Our paper focuses on the XML queries with the twig which include parent-child edges. The example of the path expression for XPath is following: bib//author [//last = 'S' AND /first = 'W'].

In the past few years, many algorithms were proposed based on twig. In particular, Bruno et al [1] propose a holistic twig joins algorithm named *TwigStack*. Using linked stacks to output partial results of individual query leaf-to-root path, *TwigStack* is I/O and CPU optimal among all twigs with only ancestor-descendant edges. But for query with parent-child edges, this algorithm may produce large useless intermediate results. To overcome this problem, Jiaheng Lu et al [2] show a novel holistic twig joins named *TwigStackList*, whose main technique is to read some elements in advance and cache them in a data structure List. *TwigStackList* is I/O and CPU optimal for queries with only ancestor-descendant edges below branching nodes and queries with parent-child edges below non-branching nodes. But it also produce intermediate results while query with parent-child edges below branching nodes. Based on the algorithm TwigStackList, Jiaheng Lu et al propose a series of twig joins algorithm, such as OrderedTJ [3] algorithm, which deal with orderedTwig efficiently, and TwigStackList¬ [4], which process XML twig queries with not-predicates.

In this paper, we present a novel twig joins algorithm PCTwigStackList. This algorithm can reduce large useless intermediate paths while we carry out query with ancestor-descendant edges and parent-child edges.

2. Notion and Data Structure

2.1. XML data encoding

XML document is always modeled as a tree. Most existing XML twig joins algorithms [5-8] use a region code (*doc*, *begin*, *end*, *level*) to present the position of an element in XML document tree. In the process of

[+] Corresponding author.
E-mail address:934198928@qq.com.

encoding, we execute preorder tree traversal over the document tree, then *begin* attribute is given before traversing ancestor element, and *end* attribute is given after traversing all descendant element, *level* is the level of the element in the document tree. The region encodings support efficient evaluation of structural relationships. Formally, we say element *u* is an ancestor of element *v* if and only if *u*. *begin* <*v*. *begin* and *v.end*<*u.end*, and for parent-child (P-C) structural relationship, it is only necessary to check whether *u.level+1=v.level*, and *doc* attribute is used for query in different XML documents.

2.2. Notion and data structure introduction

In the rest of this paper, "node" refers to a tree node in the twig pattern, and "element" refers to an element in the XML document trees.

XML query language is also presented to tree structure. The query node in twig has self-explaining functions, such as *isRoot*(*n*) or *isLeaf*(*n*) check whether a query node *n* is a root node or a leaf node. C*hildren*(*n*) returns all child nodes of *n*. A query node $n_i \in Children(n)$, if the relationship between n and n_i is parent-child, we say $n_i \in$ PC*children*(*n*) , else we say $n_i \in$ AD*children*(*n*) . So there exits two properties: $PCchildren(n) \cap ADchildren(n)=\varnothing$ and C$hildren(n) = PCchildren(n) \cup ADchildren(n)$.

Definition 1, TwigSolutionExtension: We say that a node *n* has the TwigSolutionExtension if the following three properties hold:

1. for each $n_i \in$ AD*children*(*n*), there is an element e_{ni} which is a descendant of e_n;
2. for each $n_i \in$ PC*children*(*n*) there is an element e_{ni} which is a child of e_n;
3. each $n_i \in children(n)$, n_i is also necessary to satisfy the above two conditions.

For each node in the query twig, it is associated with a data stream ***Tn*** and related data structure: a ***List*** and a ***Stack***. In the data stream *Tn*, there is a cursor ***Cn*** to point to the current retrieval element in *Tn*; and function ***end(Cn)*** examines whether *Cn* has pointed to the end of *Tn*; we can access the attribute of *Cn* by ***Cn.begin, Cn.end, Cn.level***. Initial *Cn* points to the first element of *Tn*, and we use the function ***advance (Tn)*** to jump to the next element in *Tn*.

We use *Stack* and *List* in our algorithm is similar to that in *TwigStackList*. That is, each data element in the stack has a pointer to point to top element of its parent's stack. During the query processing, *Stack* is used to cache these elements which may contribute to twig joins. All elements in *Stack* from bottom to top must guarantee follow properties:

(i) The node in stack *Sn* is an element in root to leaf path in the XML document.

(ii) The linked stacks contain all answers to the query twig pattern.

The operations on *stack* are: ***empty, pop, push, topBegin, topEnd.*** For each *List Ln*: elements are strictly nested that cache in *Ln*, each element is an ancestor of the element following it. These elements can be used to test whether they are satisfied queries edges "/" between it and its parent. *List* has an integer *pn*, which is used to locate the element's position which we retrieve in the list *Ln*. The operations over *Ln* are $L_n.empty()$, $L_n.element(pn)$, $L_n.append()$, L_n.de*lete*(*pn*) . Function $L_n.empty()$ test whether *Ln* is empty. Function $L_n.element(pn)$ get the element located by *pn*, and access its attributes through ***Ln.element(pn).begin***, ***Ln.element(pn).end*** and ***Ln.element(pn).level***. Function $L_n.append()$ append the element to the List, Function L_n.de*lete*(*pn*) delete the element located by *pn*.

3. PCTwigStackList Algorithm

3.1. PCTwigStackList algorithm description

For algorithm PCTwigStackList, we describe the detail in Algorithm 2. This algorithm operates in two phases. In the first phase (Algorithm 2 1ine 1~10), the main algorithm repeatedly calls its subroutine *getNext* to get the next element for twig joins processing. We output the partial solutions with root-to-leaf paths in this phase. In the second phase (line 12), these solutions are merged to form all the answers to the query twig pattern.

Subroutine *getNext* is the core operation in twig joins algorithm, which is defined in Algorithm 1. We can skip the elements that they are impossible to participate in twig joins, and return the element which is most possible to participate in final joins to the main algorithm.

Algorithm 1 getNext (n)

```
 1: if isLeaf(n)
 2:       if (Ln.isEmpty) then
 3:             do append Cn to Ln and Cn.Anvance
 4:          while Cn is descendant of the tail element of Ln
 5:       return n
 6: for all node ni in Children(n) do
 7:   gi=getNext (ni)
 8:   if(gi≠ni) return gi
 9: nmax=maxarg { ni∈ children(n)},    nmin=minarg { ni∈ children(n)}
10:      while (!endof() )
11:      while (getEnd(n)<getBegin(nmax)) proceed(n)
12:      if (getBegin(n)>getBegin(nmin)) return nmin
13:   MoveStreamToList(n, nmax)
14:      for all node ni in PCchildren(n) do
15:            if (there is an element ei in Ln such that ei is the parent of getElement(ni) then
16:            if (ni is the only child of n)
17:                        move the cursor Pn of list to point to ei
18:            else return ni
19:    if getElement(n) is not the parent of getElement(ni) and not the ancestor of Cni then
             proceed(n)
20:            return n
Function getElement(n)
       if ¬Ln.empty then return Ln.elementAt(pn)
       else Cn
Function getBegin(n) return begin attribute of getElement(n)
Function getEnd(n) return end attribute of getElement(n)
Function proceed(n)
       if ¬Ln.empty then
            delete Ln.elementAt(pn) and pn = 0
       else Cn.Advance
Procedure MoveStreamToList(n , max)
       while Cn.begin<getBegin(max) do
            if (Cn.end<getEnd(max) then append Cn to Ln
            advance(Cn)
```

At line 1~5 in Algorithm 1, this is different from *TwigStackList*, we must read a group strictly nested elements in advance and cache them in *List*. At line 6~8, we recursively call *getNext* for each $n_i \in children(n)$. If any returned element *gi* is not equal to *ni*, we return *gi* (line 8) immediately.

At line 10~20, this ***while*** loop is a very important query step, and its major processing is to distinguish the elements whether they can contribute to twig joins. Line 11 skips elements that do not contribute to results. If there is no common ancestor for all children, line 12 returns the smallest child node. At line 13, we read a group of elements from the stream *Tn* and store them in *List Ln* that there are ancestors of Cn_{max}. In query processing, we can check whether there is an elements in *Ln* that it is the parent of *getElement(n)*. Line 19, this step can skip more elements that we can sure they are not results of twig.

Lemma 1: It is impossible for the elements skipped at line 19 to take part in twig joins.

Proof: because elements in *Ln* is strictly nested, if getElement(n) is not the parent of getElement(n_i), in other word, getElement(n_i) is ancestor of other elements in L_{ni}, we can make sure that getElement(n) is not the parent of other elements in L_{ni}. And we know that getElement(n) is not ancestor of C_{ni} from given condition. So getElement(n) is not a parent of any element.

In other previous twig joins algorithms, such as *TwigStackList.* The main algorithm output all individual paths matching from leaf to root when the subroutine *getNext* returns a leaf element and caches it in *Stack*.

The reason of leaf to root is a sub query of twig, and this step always produces large number useless paths. In order to reduce these useless partial paths, our technique is to delay outputting the individual path matches in *PCTwigStackList*. We keep these elements in its *Stack* before they encounter a new element that they must be cleaned out and pop from stack under certain condition. In this situation, we output all partial solutions from leaf to root. The detailed description is as follow: before clean stack, we check the elements in *Stack* whether they have TwigSolutionExtension, if they don't have TwigSolutionExtension, we can pop these elements from stack safely, and else we must output the all partial results before clean the stacks as previous algorithms.

The function of LeafToRootSolution(n) is to get all leaves of sub twig that the root node is n, and output all path matching from these leaves to root.

Algorithm 2 PCTwigStackList

```
1 while (! endof ())
2  n_act=getNext (root)
3  if (¯isRoot(n_act))  then
4    cleanStack (parent (n_act), getEnd(n_act) )
5    moveToStack(n_act, Sn_act, pointertoTop(Sparent(n_act))
6  if (isRoot(n_act) || ¯parent (n_act).isEmpty) then
7    cleanStack (n_act , getEnd(n_act) )
8    moveToStack(n_act, Snact, pointertoTop(Sparent(n_act))
9  else  proceed(n_act)
10  LeafToRootSolution(root)
11  mergeAllPathSolutions()
Procedure  cleanStack (n , getEnd (n_act))
1: while (¯empty(S_n) && topEnd(S_n)<getEnd(n_act))
2:        if top(S_n) is not a TwigSolutionExtension then
3:                modify related Stack element pointer point to parent
3:                pop(S_n)
3:        else break;
4:       if(¯empty(S_n)&& topEnd(S_n)<getEnd(n_act))
5:          LeafToRootSolution(n)
6: while (¯empty(S_n)&& topEnd(S_n)<getEnd(n_act))
7:         pop(S_n)
Procedure  LeafToRootSolution(n)
1:       if  ¯isLeaf(n)
2:                for every child n_i of n
3:                        LeafToRootSolution(n_i)
4:       else
5:       showSolutionsWithBlocking(top(S_n))
6:                pop(S_n)
```

The function of cleanStack (n, getEnd (n_{act})) is used to pop all elements which do not participate in any new solution, and output individual path matches. We sure that if the coming element is satisfied this condition of "topEnd(S_n)<getEnd(n_{act})", we can pop the top element in Stack safely. In TwigStackList, it is divided into parent stack and self stack. We can merge them into one condition.

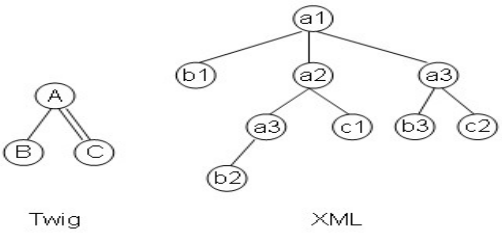

Fig. 1: XML document and Twig

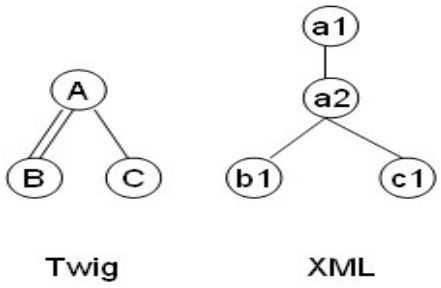

Fig. 2: XML document and Twig

Example 1: Consider in Fig.1. In *TwigStackList* algorithm, it produces a useless path of c1-a2. But in our algorithm, *getNext* return with a3, we check the top element in stack (a2) doesn't have TwigSolutionExtension, so we pop a2. After this process, our algorithm does not output useless path maching c1-a2.

Our algorithm produce less useless path matches compare to *TwigStackList*. But it also not optimal for I/O and CPU while query twig mix ancestor-descendant edge and parent-child edge.

Example 2: Consider in Fig.2. Our algorithm will produce the useless path b1-a1.

4. Experimental Evaluation

In order to evaluate the performance of our algorithm, we show experimental results in this section. We implement three algorithms: *TwigStack*, *TwigStackList*, *PCTwigStackList* in java programming language with the open source tools Eclipse. The XML parser we used is dom4j. All experiments were performed on a PC with Pentium(R) 4 CPU 2.66GHz and 768MB RAM. The Operating System is Windows XP. We used the two data sets for evaluation: TreeBank and DBLP[9]. The information of data sets is as follow:

Treebank: size(82M), elements(2437666), attributes (1), max-depth(36).

DBLP: size(127M),elements(3332130), attributes (404276), max-depth(6).

While the processing of experiments, we use two metrics:

4.1. The total number of partial path solutions

It can reflect the ability of algorithms to reduce the size of intermediate results for different kinds of twigs.

4.2. Running time

The running time of an algorithm is the core metrics of algorithms. It reflects the efficiency of an algorithm. It should be obtained through several runs the programming, and calculate the average of the running time.

The queries on both data sets are showed in Tab.1. We can know that all queries in our experiment are different compared with each other. They can show the performance of all algorithms in experiment.

Table. 1: XML data set and XPath

Query ID	Data set	XPath expression
Q1	TreeBank	//S[/JJ]/NP
Q2	TreeBank	//S/VP/PP[/NP/VBN]/IN
Q3	TreeBank	//S[//VP/VBD]/CC
Q4	TreeBank	//VP[/PP//NNP]//JJ
Q5	TreeBank	//S/VP[//NN]/VBD
Q7	DBLP	//inproceedings/title[//i]/sup
Q7	DBLP	//article[//sup]//title/sub
Q8	DBLP	//dblp/inproceedings[//cite/label]title
Q9	DBLP	/dblp/inproceedings[title]/author
Q10	DBLP	//article[//volume][//cite]//journal

The queries over Treebank, The metrics of total numbers of partial path solutions are presented in Tab.2. We can know the PCTwigStackList can reduce the size of intermediate results efficiently.

Table. 2: The count of path

Algorithm	Q1	Q2	Q3	Q4	Q5
TwigStack	70988	2237	29704	44136	99320
TwigStackList	30	388	9153	32460	97203
PCTwigStackList	21	353	7534	27361	88307

The experimental results are showed in Fig.3 and Fig.4. As shown, while our queries mix ancestor-descendant edge and parent-child edge, *PCTwigStackList* has the better performance than *TwigStackList* and *TwigStack*.

5. Summery

We propose a holistic twig join algorithm that retrieves root to path matches efficiently. Our algorithm can avoids much more useless intermediate path matches for twig patterns. The better performance of our algorithm has been substantiated in our experiments.

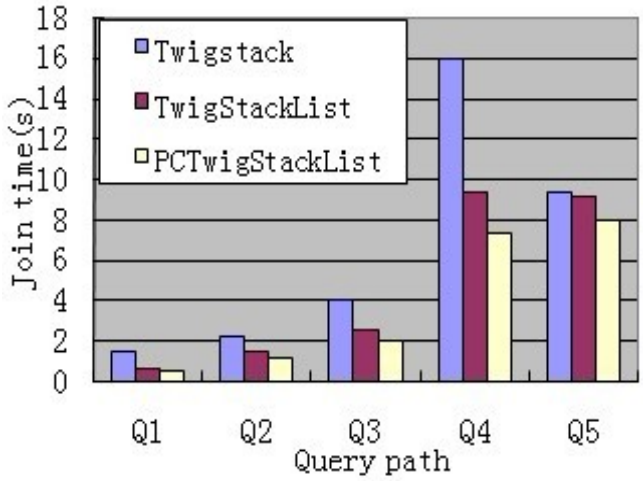

Fig. 3: XML queries over Treebank

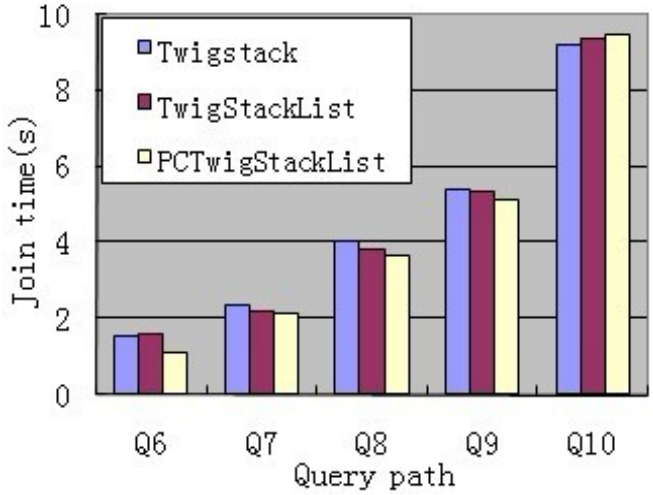

Fig. 4: XML queries over DBLP

6. Acknowledgements

This work is sponsored by 2009 Open Research Funds, Key Lab of Spatial Data Mining & Information Sharing, Ministry of Education under Grand No. 210006

7. References

[1] N.Bruno, D.Srivastava and N.Koudas. Holistic twig joins: Optimal XML pattern matching. In SIGMOD, pages 310-321,2002.

[2] L.Jiaheng, C.Ting and L.Tok Wang. Efficient Processing of XML Twig Patterns with Parent Child Edges :A Look-ahead Approach. 2004 In Proceedings of CIKM, pp.533-542.

[3] J. Lu, T. W. Ling, T. Yu, C. Li et al. Efficient processing of ordered XML twig pattern matching, Proceedings of DEXA, 16th International Conference, 2005, pp. 300–309.

[4] Tian Yu, Tok Wang,Wang Ling et al. TwigStackList¬: A Holistic Twig Join Algorithm for Query with Not-Predicates on XML Data. In DASFAA ,2006 LNCS 3882,pp. 249–263.

[5] Jiang Li and Junhu Wang. TwigBuffer: Avoiding Useless Intermediate Solutions Completely in Twig Joins. In DASFAA 2008, LNCS 4947, pp. 554–561.

[6] Chen T,Lu J ,Ling T W et al. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In:Proceedings of the ACM SIGMOD International Conference on Management of Data. Baltimore,Maryland. June 14-16,2005. ACM Press. 455~466.

[7] Zhewei J,Cheng L,Wen-Chi et al. Efficient of XML Twig Pattern: A Novel one-phase Holistic Solution. In DEXA 2007,LNCS 4653,pp.87-97.

[8] Guoliang Li, Jianhua Feng, Yong Zhang et al. “Efficient Holistic Twig Joins in Leaf-to-Root Combiningwith Root-to-Leaf Way”,In DASFAA, 2007, page 834-849.

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