Efficient Processing of Ordered XML Twig Pattern

Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, and Wei Ni

School of computing, National University of Singapore
{lujiahen, lingtw, yutian, lichangq, niwei}@comp.nus.edu.sg

Abstract. Finding all the occurrences of a twig pattern in an XML database is a core operation for efficient evaluation of XML queries. Holistic twig join algorithm has showed its superiority over binary decompose based approach due to efficient reducing intermediate results. The existing holistic join algorithms, however, cannot deal with *ordered* twig queries. A straightforward approach that first matches the unordered twig queries and then prunes away the undesired answers is obviously not optimal in most cases. In this paper, we study a novel holistic-processing algorithm, called *OrderedTJ*, for ordered twig queries. We show that *OrderedTJ* can identify a large query class to guarantee the I/O optimality. Finally, our experiments show the effectiveness, scalability and efficiency of our proposed algorithm.

1 Introduction

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a tree-structured data model([1],[5]). Efficient finding all twig patterns in an XML database is a major concern of XML query processing. Recently, holistic twig join approach has been taken as an efficient way to match twig pattern since this approach can efficiently control the size of intermediate results([1],[2],[3],[4]). We observe that, however, the existing work on holistic twig query matching only considered *unordered* twig queries. But XPath defines four ordered axes: *following-sibling*, *preceding-sibling*, *following*, *preceding*. For example, XPath: *//book/text/following-sibling::chapter* is an ordered query, which finds all *chapters* in the dataset that are following siblings of *text* which should be a child of *book.*

We call a twig query which cares the order of the matching elements as an *ordered* twig query. On the other hand, we denote a twig query that does not consider the order of matching elements as an *unordered* query. In this paper, we research how to efficiently evaluate an *ordered* twig query.

To handle an ordered twig query, naively, we can use the existing algorithm (e.g. TwigStack[1]/TwigStackList[5]) to output the intermediate path solutions for each individual root-leaf query path, and then merge path solutions so that the final solutions are guaranteed to satisfy the order predicates of the query. Although existing algorithms are applied, such a post-processing approach has a serious disadvantage: many intermediate results may not contribute to final answers.

K.V. Andersen, J. Debenham, and R. Wagner (Eds.): DEXA 2005, LNCS 3588, pp. 300 – 309, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Motivated by the recent success in efficient processing unordered twig queries *holistically*, we present in this paper a novel holistic algorithm, called *OrderedTJ*, for *ordered* twig queries. The contribution of this paper can be summarized as follows:

1. We develop a new holistic ordered twig join algorithm, namely *OrderedTJ,* based on the new concept of *Ordered Children Extension (for short OCE).* With OCE, an element contributes to final results only if the order of its children accords with the order of corresponding query nodes. Thus, efficient holistic algorithm for ordered-twigs can be leveraged.
2. If we call edges between branching nodes and their children as *branching edges* and denote the branching edge connecting to the *n*'th child as the *n'th branching edge*, we analytically demonstrate that when the ordered-twig contains only ancestor-descendant relationship from the 2nd branching edge, *OrderedTJ* is I/O optimal among all sequential algorithms that read the entire input. In other words, the optimality of *OrderedTJ* allows the existence of *parent-child* relationships in *non-branching* edges and the *first* branching edges.
3. Our experimental results show that the effectiveness, scalability and efficiency of our holistic twig algorithms for ordered twig pattern.

The remainder of the paper is organized as follows. Section 2 presented related work. The novel ordered twig join algorithm is described in Section 3. Section 4 is dedicated to our experimental results and we close this paper by conclusion and future work in Section 5.

2 Related Work

With the increasing popularity of XML data, query processing and optimization for XML databases have attracted a lot of research interest. There is a rich set of literature on matching twig queries efficiently. Below, we describe these literatures with the notice that the existing work deals with only *unordered* twig queries.

Zhang et al.([9]) proposed a multi-predicate merge join (MPMGJN) algorithm based on (*DocId*, *Start*, *End*, *Level*) labeling of XML elements. The later work by Al-Khalifa et al.([7]) gave a stack-based binary structural join algorithm. Different from binary structural join approaches, Bruno et al.([1]) proposed a holistic twig join algorithm, called *TwigStack*, to avoid producing a large intermediate result. However, the class of optimal queries in *TwigStack* is very small. When a twig query contains any *parent-child* edge, the size of "*useless"* intermediate results may be very large. Lu et al.([5]) propose a new algorithm called *TwigStackList*. They use *list* data structure to cache limited elements to identify a larger optimal query class. *TwigStackList* is I/O optimal for queries with only *ancestor-descendant* relationships in all branching edges. Recently, Jiang et al.([3]) researched the problem of efficient evaluation of twig queries with OR predicates. Chen et al.([2]) researched the relationship between different data partition strategies and the optimal query classes for holistic twig join. Lu et al.([6]) proposed a new labeling scheme called *extended Dewey* to efficiently process XML twig pattern.

3 Ordered Twig Join Algorithm

3.1 Data Model and Ordered Twig Pattern

We model XML documents as *ordered* trees. Figure 1(e) shows an example XML data tree. Each tree element is assigned a region code (*start*, *end*, *level*) based on its position. Each text is assigned a region code that has the same *start* and *end* values.

XML queries make use of twig patterns to match relevant portions of data in an XML database. The pattern edges are parent-child or ancestor-descendant relationships. Given an *ordered* twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from the nodes in Q to the elements in D, such that: (i) the query node predicates are satisfied by the corresponding database elements; and (ii) the parent-child and ancestor-descendant relationships between query nodes are satisfied by the corresponding database elements; and (iii) the *orders* of query sibling nodes are satisfied by the corresponding database elements. In particular, with region encoding, given any node $q \in Q$ and its right-sibling $r \in Q$ (if any), their corresponding database elements, say e_q and e_r in D, must satisfy that $e_q.end<e_r.start$.

The answers to query Q with *n* nodes can be represented as a list of *n-ary* tuples, where each tuple $(t_1,t_2,\dots.t_n)$ consists of the database elements that identify a distinct match of Q in D.

Figure 1(a) shows three sample XPath and Figure 1(b-d) shows the corresponding ordered twig patterns for the data of Fig 1(e). For each branching node, we use a symbol “>” in a box to mark its children ordered. Note that in Q3, we add *book* as the root of the ordered query, since it is the *root* of XML document tree. For example, the query solution for Q3 is only <$book_1$, $chpater_2$, $title_2$, “*related work*”, $section_3$ >. But if Q3 were an *unordered* query, $section_1$, $section_2$ also would involve in answers.

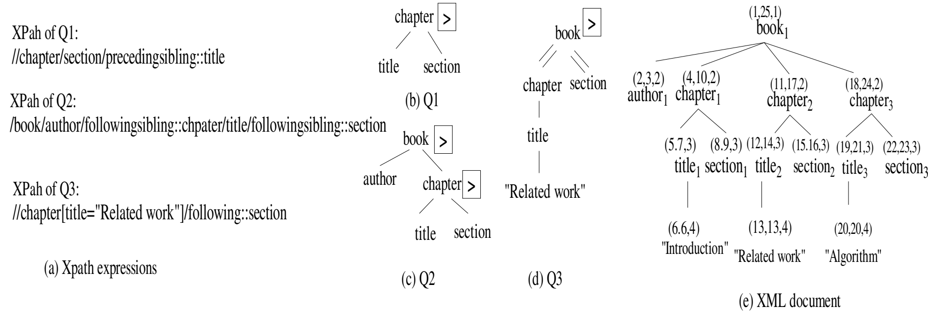


Fig. 1. (a) three XPaths (b)-(d) the corresponding ordered twig query (e) an XML tree

3.2 Algorithm

In this section, we present *OrderedTJ*, a novel holistic algorithm for finding all matches of an *ordered* twig pattern against an XML document. *OrderedTJ* makes the extension of *TwigStackList* algorithm in the previous work [5] to handle *ordered* twig pattern. We will first introduce data structures and notations to be used by *OrderedTJ*.

Notation and data structures. An ordered query is represented with an *ordered* tree. The function PCRchildren(n) , ADRChildren(n) return child nodes which has parent-child or ancestor-descedant relationships with *n,* respectively. The self-explaining function rightSibling(n) returns the immediate right sibling node of *n* (if any).

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. We can access the values of C_n by C_n.start,C_n.end and C_n.level. The cursor can advance to the 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 use two types of data structures: *list* and *stack*. We associate a list L_n and a stack S_n for each node of queries. At every point during computation: the nodes in stack S_n are guaranteed to lie on a root-leaf path in the database. We use top(S_n) to denote the top element in stack S_n. Similarly, elements in each list L_n are also strictly nested from the first to the end, i.e. each element is an ancestor or parent of that following it. For each list L_n, we declare an integer variable, say p_n, as a cursor to point to an element in L_n. Initially, p_n =0 , which points to the first element of L_n.

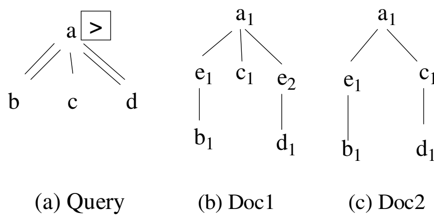


Fig. 2. Illustration to ordered child extension

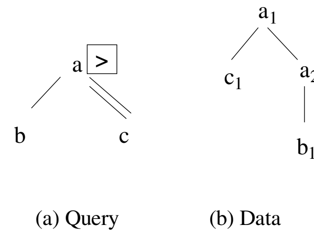


Fig. 3. Optimality example

The challenge to ordered twig evaluation is that, even if an element satisfies the parent-child or ancestor-descendant relationship, it may not satisfy the order predicate. We introduce a new concept, namely *Ordered Children Extension* (for short, OCE), which is important to determine whether an element likely involves in ordered queries.

DEFINITION 1(**OCE**) *Given an ordered query Q and a dataset D, we say that an element e_n (with tag $n \in Q$) in D has an ordered children extension (for short OCE), if the following properties are satisfied:*

(i) *for $n_i \in ADRChildren(n)$ in Q (if any) , there is an element e_{n_i} (with tag n_i) in D such that e_{n_i} is a descendant of e_n and e_{n_i} also has OCE;*

(ii) *for $n_i \in PCRChildren(n)$ in Q (if any), there is an element e' (with tag n) in the path e_n to e_{n_i} such that e' is the parent of e_{n_i} and e_{n_i} also has OCE;*

(iii) *for each child n_i of n and m_i= rightSibling(n_i) (if any), there are elements e_{n_i} and e_{m_i} such that e_{n_i}.end < e_{m_i}.start and both e_{n_i} and e_{m_i} have OCE.* □

Properties (i) and (ii) discuss the ancestor-descendant and parent-child relationship respectively. Property (iii) manifests the order condition of queries. For example, see

the ordered query in Fig 2(a), in Doc 1, a_1 has the OCE, since a_1 has descendants b_1,d_1, child c_1 and more importantly, b_1,c_1,d_1 appear in the correct order. In contrast, in Doc2, a_1 has **not** the OCE, since d_1 is the descendant of c_1, but not the ***following*** element of c_1(i.e. c_1.end $\not<$ d_1 .start).

```
Algorithm OrderedTJ ()
01While (¬end())
02     q_act= getNext(root);
03     if (isRoot(q_act) ∨ ¬empty(S_parent(q_act))) cleanStack(q_act, getEnd(q_act));
04     moveStreamToStack(q_act, S_q_act );
05     if (isLeaf(q_act))
06              showpathsolutions (S_q_act ,getElement(q_act));
07     else       proceed(T_q_act );
08 mergeALLPathsolutions;

Function end()
01 return   ∀n ∈ subtreesNodes(root) : isLeaf(n) ∧ eof(C_n) ;

Procedure cleanStack (n ,actEnd)
01 while (¬empty(S_n) and (topEnd(S_n)< actEnd)) do pop(S_n);

Procedure moveStreamToStack(n,S_n )
01 if((getEnd(n)< top(S_rightsibling(n)).start) //check order
02     push getElement(n) to stack S_n
06     proceed(n);

Procedure proceed(n)
01 if (empty(L_n))  advance(T_n);
02 else  L_n.delete(p_n);
03      p_n = 0; //move p_n to pint to the beginning of L_n

Procedure showpathsolutions(S_m,e)
01 index[m]=e
02 if (m == root ) //we are in root
03  Output(index[q_1],…,index[q_k]) //k is the length of path processed
04 else  //recursive call
05   for each element e_i in S_parent(m)
06        if  e_i satisfies the corresponding relationship with e
07                    showpathsolutions(S_parent(m), e_i)
```

Fig. 4. Rocedure OrderedTJ

Algorithm OrderedTJ. *OrderedTJ*, which computes answers to an ordered query twig, operates in two phases. In the first phase (line 1-7), the individual query root-leaf paths are output. In the second phase (line 8), these solutions are merged-joined to compute the answers to the whole query. Next, we first explain *getNext* algorithm which is a core function and then presents the main algorithm in details.

getNext(n)(See Fig 5) is a procedure called in the main algorithm of *OrderedTJ.* It identifies the next stream to be processed and advanced. At line 4-8, we check the condition (i) of OCE. Note that unlike the previous algorithm *TwigStackList[5]*, in line 8, we advance the **maximal** (not minimal) element that are not descendants of the

current element in stream T_n , as we will use it to determine sibling order. Line 9-12 check the condition (iii) of OCE. Line 11 and 12 return the elements which violate the query sibling order. Finally, line 13-19 check the condition (ii) of OCE.

Now we discuss the main algorithm of *OrderedTJ*. First of all, Line 2 calls *getNext* algorithm to identify the next element to be processed. Line 3 removes partial answers that cannot be extended to total answer from the stack. In line 4, when we insert a new element to stack, we need to check whether it has the appropriate right sibling. If *n* is a leaf node, we output the whole path solution in line 6.

```
Algorithm getNext (n)
01  if (isLeaf(n))  return n;
02  for all n_i in children(n)  do
03        g_i = getNext(n_i);   if (g_i ≠ n_i ) return n_i ;
04  n_max = max arg_{n_i ∈ children(n)} getStart(n_i);
05  n_min = min arg_{n_i ∈ children(n)} getStart(n_i);
06  While (getEnd(n) < getStart(n_max))  proceed(n);
07  if (getStart(n) >getStart(n_min))
08        return   max arg_{n_i ∈ children(n) ∧ (getStart(n) > getStart(n_i))} getStart(n_i);
09   sort all n_i in children(n) by start values;
      // assume the new order are n'_1,n'_2, …,n'_k
10  for each n'_i   (1 ≤ i ≤  n)  do   //check children order
11        if (n'_i ≠ n_i)  return  n'_i;
12        else if ((i>1)∧(getEnd(n'_{i-1})>getStart(n'_i)) return n'_{i-1}
13  MoveStreamToList(n, n_max);
14  for n_i in PCRchildren(n)  //check parent-child relationship
15    if (∃e'∈ L_n  such that e' is the parent of  C_{n_i} )
16        if  (n_i is the first child of n)
17            Move the cursor of list L_q to point to e';
18        else  return n_i ;
19  return n;

Proceudre MoveStreamToList(n,g)
01 delete any element in L_n  that is not an ancestor of getElement(n);
02 while C_n.start < getStart(g) do  if C_n.end>getEnd(g)   L_n.append(C_n);
03                                    advance(T_n)

Procedure getElement(n)
01  if (¬empty(L_n))       return L_n.elementAt(p_n);
02  else   return C_n;

Procedure getStart(n)
01 return the start attribute of getElement(n);

Procedure getEnd(n)
01 return the end attribute of getElement(n);
```

Fig. 5. Function GetNext in the main algorithm OrderedTJ

EXAMPLE 1. *Consider the ordered query and data in Fig 1(d) and (e) again. First of all, the five cursors are (book$_1$, chapter$_1$, title$_1$,"related work", section$_1$). After two calls of* ***getNext****(book), the cursors are forwarded to (book$_1$, chapter$_2$, title$_2$, "related work", section$_1$). Since* section$_1$.*start=6<*chapter$_2$.*start=9, we return* section *(in line 11 of getNext) and forward to* section$_2$. *Then* chapter$_2$.*end=15> section$_2$.start=13. We*

return section *again (in line 12 of getNext) and forward to* $section_3$. *Then chapter*$_2$*.end=15<section*$_3$*.start=17. The following steps push* $book_1$ *to stack and output the individual two path solutions. Finally, in the second phase of main algorithm, two path solutions are merged to form one final answer* □

3.3 Analysis of OrderedTJ

In the section, we show the correctness of *OrderedTJ* and analyze its efficiency. Some proofs are omitted here due to space limitation.

DEFINITION 2 (**head element** e_n) *In OrderedTJ, for each node in the ordered query, if List* L_n *is not empty, then we say that the element indicated by the cursor* p_n *of* L_n *is the head element of n, denoted by* e_n*. Otherwise, we say that element* C_n *in the stream* T_n *is the head element of n.* □

LEMMA 1. *Suppose that for an arbitrary node n in the ordered query we have getNext(n)=n'. Then the following properties hold:*

(1) *n' has the OCE.*
(2) *Either (a) n=n' or (b) parent(n) does not have the OCE because of n' (and possibly a descendant of n').*

LEMMA 2. *Suppose getNext(n)=n' returns a query node in the line 11 or 12 of Algorithm getNext. If the current stack is empty, the head element does not contribute to any final solution since it does not satisfy the order condition of query.*

LEMMA 3. *In Procedure moveStreamToStack any element e that is inserted to stack* S_n *satisfy the order requirement of the query. That is, if n has a right-sibling node n' in query, then there is an element* e_n*' in stream* T_n*' such that* e_n*'.start >*e_n*.end.*

LEMMA 4. *In OrderedTJ, when any element e is popped from stack , e is guaranteed not to participate a new solution any longer.*

THEOREM 1. *Given an ordered twig pattern Q and an XML database D. Algorithm OrderedTJ correctly returns all answers for Q on D.*

Proof*:[sketch]* Using Lemma 2, we know that when *getNext* returns a query node *n* in the line 11 and 12 of *getNext*, if the stack is empty, the head element e_n does not contribute to any final solutions. Thus, any element in the ancestors of *n* that use e_n in the OCE is returned by the *getNext* before e_n. By using lemma 3, we guarantee that each element in stack satisfy the order requirement in the query. Further. By using lemma 4, we can maintain that, for each node n in the query, the elements that involve in the root-leaf path solution in the stack S_n. Finally, each time that *n =getNext(root)* is a leaf node, we output all solution for e_n (line 6 of *OrderedTJ*). □

Now we analyze the optimality of *OrderedTJ*. Recall that the unordered twig join algorithm *TwigStackList*([5]) is optimal for query with only ancestor-descendant in all branching edges, but our *OrderedTJ* can identify a little **larger** optimal class than *TwigStackList* for ordered query. In particular, the optimality of *OrderedTJ* allows the existence of parent-child relationship in the **first** branching edge, as illustrated below.

EXAMPLE 2. *Consider the ordered query and dataset in Fig 3. If the query were an unordered query, then* TwigStackList([5]) *would scan* a_1, c_1 *and* b_1 *and output one useless solution (*a_1,c_1*), since before we advance* b_1 *we could not decide whether* a_1 *has a child tagged with* b. *But since this is an ordered query, we immediately identify that* c_1 *does not contribute to any final answer since there is no element with name* b *before* c_1. *Thus, this example tells us that unlike algorithms for unordered query,* OrderedTJ *may guarantee the optimality for queries with parent-child relationship in the first branching edge.* □

THEOREM 2. *Consider an XML database D and an ordered twig query Q with only ancestor-descendant relationships in the n'th (n≥2) branching edge. The worst case I/O complexity of* OrdereTJ *is linear in the sum of the sizes of input and output lists. The worst-case space complexity of this algorithm is that the number of nodes in Q times the length of the longest path in D.* □

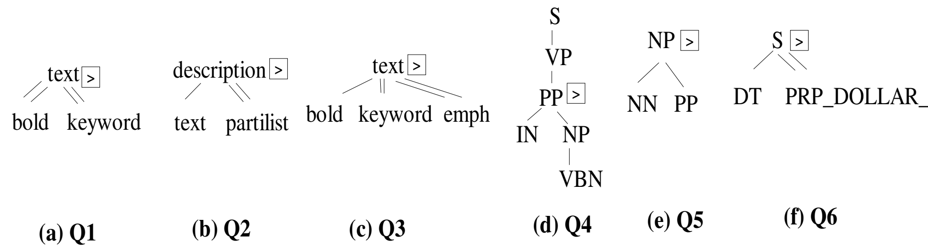


Fig. 6. Six tested ordered twig queries (Q1,2,3 in XMark; Q4,5,6 in TreeBank)

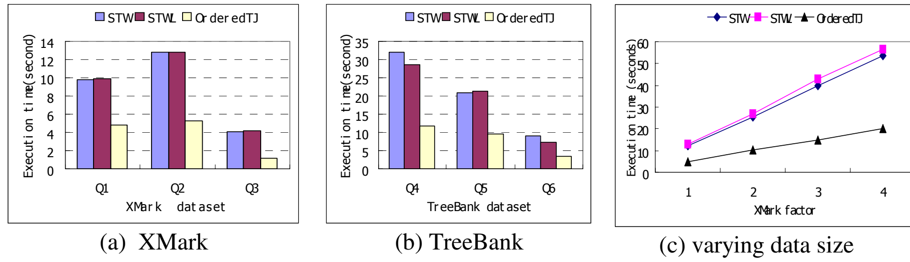
4 Experimental Evaluation

4.1 Experimental Setup

We implemented three ordered twig join algorithms: *straightforward -TwigStack (for short STW)*, *straightforward-TwigStackList* (*STWL*) and *OrderedTJ*. The first two algorithms use the straightforward post-processing approach. By post-processing, we mean that the query is first matched as an unordered twig (by TwigStack[1] and TwigStackList[5], respectively) and then we merge all intermediate path solutions to get the answers for an ordered twig. We use JDK 1.4 with the file system as a simple storage engine. All experiments were run on a 1.7G Pentium IV processor with 768MB of main memory and 2GB quota of disk space, running windows XP system. We used two data sets for our experiments. The first is the well-known benchmark data: XMark. The size of file is 115M bytes with factor 1.0. The second is a real data-set: TreeBank[8]. The deep recursive structure of this data set makes this an interesting case for our experiments. The file size is 82M bytes with 2.4 million nodes.

For each data set, we tested three XML twig queries (see Fig 6). These queries have different structures and combinations of parent-child and ancestor-descendant edges. We choose these queries to give a comprehensive comparison of algorithms.

Evaluation metrics. We will use the following metrics to compare the performance of different algorithms. (i) **Number of intermediate path solutions** This metric measures the total number of intermediate path solutions, which reflects the ability of algorithms to control the size of intermediate results. (ii) **Total running time** This metric is obtained by averaging the total time elapsed to answer a query with six consecutive runs and the best and worst performance results discarded.



(a) XMark (b) TreeBank (c) varying data size

Fig. 7. Evaluation of ordered twig pattern on two datasets

Table 1. The number of intermediate path solutions

Query	Dataset	STW	STWL	OrderedTJ	Useful solutions
Q1	XMark	71956	71956	44382	44382
Q2	XMark	65940	65940	10679	10679
Q3	XMark	71522	71522	23959	23959
Q4	TreeBank	2237	1502	381	302
Q5	TreeBank	92705	92705	83635	79941
Q6	TreeBank	10663	11	5	5

4.2 Performance Analysis

Figure 7 shows the results on execution time. An immediate observation from the figure is that *OrderedTJ* is more efficient than STW and STWL for all queries. This can be explained that *OrderedTJ* output much less intermediate results. Table 1 shows the number of intermediate path solutions. The last column shows the number of path solutions that contribute to final solutions. For example, STW and STWL could output 500% more intermediate results than *OrderedTJ* (see XMark Q2).

Scalability. We tested queries XMark Q2 for scalability. We use XMark factor 1(115MB)， 2(232MB)，3 (349M) and 4(465M). As shown in Fig 7(c), *OrderedTJ* scales linearly with the size of the database. With the increase of data size, the benefit of *OrderedTJ* over STW and STWL correspondingly increases.

Sub-optimality of OrderedTJ. As explained in Section 3, when there is any parent-child relationship in the *n'th branching edges* (n≥2), *OrderedTJ* is not optimal. As shown in Q4,Q5 of Table 1, none of algorithms is optimal, since all algorithms output some useless solutions. However, even in this case, *OrderedTJ* still outperforms STW and STWL by outputting **less** useless intermediate results.

Summary. According to the experimental results, we draw two conclusions. First, our new algorithm *OrderedTJ*, could be used to evaluate ordered twig pattern because they have obvious performance advantage over the straightforward approach: STW and STWL. Second, *OrderedTJ* guarantee the I/O optimality for a large query class.

5 Conclusion and Future Work

In this paper, we proposed a new holistic twig join algorithm, called *OrderedTJ*, for processing **ordered** twig query. Although the idea of *holistic* twig join has been proposed in unordered twig join, applying it for ordered twig matching is nontrivial. We developed a new concept *ordered child extension* to determine whether an element possibly involves in query answers. We also make the contribution by identifying a large query class to guarantee I/O optimal for *OrderedTJ*. Experimental results showed the effectiveness, scalability, and efficiency of our algorithm.

There is more to answer XPath query than is within the scope of this paper. Consider an XPath query: "//a/following-sibling::b", we cannot transform this query to an ordered twig pattern, since there is no *root* node in this query. Thus, algorithm *OrderedTJ* cannot be used to answer this XPath. In fact, based on region code (*start*,*end*,*level*), none of algorithms can answer this query by accessing the labels of *a* and *b* alone, since *a* and *b* may have *no common parent* even if they belong to the same level. We are currently designing a new labeling scheme to handle such case.

References

1. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML pattern matching. In *Proc. of the SIGMOD, pages 310-321 2002*.
2. T. Chen J. Lu , and T. W Ling On boosting holism in XML twig pattern matching using structural indexing techniques In *Proc. of the SIGMOD* 2005 To appear
3. H. Jiang, H. Lu, W. Wang, Efficient Processing of XML Twig Queries with OR-Predicates, In *Proc. of the SIGMOD pages 59-70 2004*.
4. H. Jiang, et al. Holistic twig joins on indexed XML documents. In *Proc. of the VLDB, pages 273-284, 2003*.
5. J. Lu , T. Chen and T. W. Ling Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach *In Proc. of CIKM, pages 533-542, 2004*
6. J. Lu et. al From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching *In Proc. of VLDB, 2005* To appear
7. S. Al-Khalifa et. al Structural joins: A primitive for efficient XML query pattern matching. *In Proc. of the ICDE, pages 141-152, 2002.*
8. Treebank http://www.cs.washington.edu/research/xmldatasets/www/repository.html
9. C. Zhang et. al. On supporting containment queries in relational database management systems. *In Proc. of the SIGMOD, 2001.*