# Efficient Processing of XML Twig Pattern : A Novel One-Phase Holistic Solution

Zhewei Jiang[1], Cheng Luo[1], Wen-Chi Hou[1], Qiang Zhu[2], and Dunren Che[1]

[1] Computer Science Department, Southern Illinois University Carbondale, Carbondale, IL 62901, U.S.A.
{zjiang, cluo, hou, dche}@cs.siu.edu
[2] Department of Computer and Info. Science, University of Michigan Dearborn, MI, 48128, U.S.A.
qzhu@umich.edu

**Abstract.** Modern twig query evaluation algorithms usually first generate individual path matches and then stitch them together (through a "merge" operation) to form twig matches. In this paper, we propose a one-phase holistic twig evaluation algorithm based on the TwigStack algorithm. The proposed method applies a novel stack structure to preserve the holisticity of the twig matches. Without generating intermediate path matches, our method avoids the storage of individual path matches and the path merge process. Experimental results confirm the advantages of our approach.

## 1 Introduction

XML has become a widely accepted standard for data exchange and integration over the Internet. The ability to process XML queries efficiently plays an important role in the deployment of the XML technology in the future.

The XML twig queries retrieve document elements through a joint evaluation of multiple path expressions [8]. Modern XML twig query processing approaches [1, 9, 2, 6, 4], including the Twigstack [2] and other approaches based on it, typically first decompose a twig query into a set of binary patterns or single paths and then search for matches for these individual patterns/paths. Finally, these matches are stitched together to form the answers to the twig query. The overheads incurred in the two-phase approach could be large since the cost to output and then input the individual matches and finally merge them to form twig matches can be substantial, especially when the number of matching paths is large.

To address this problem, we propose a one-phase holistic twig evaluation algorithm that outputs twig matches in their entireties without a later merge process. Instead of outputting individual path matches as soon as they are formed, our method holds the path matches until entire twig matches are formed. The new algorithm yields no intermediate results (to be output and then input), and requires no additional merge phase. Experimental results show that our algorithm compares favorable to Twigstack [2] and Twig2Stack [11].

The rest of the paper is organized as follows. Section 2 gives a brief survey of twig processing. Section 3 details our one-phase holistic twig evaluation algorithm. Section 4 presents the experimental results. Section 5 concludes.

## 2 Backgrounds

Many twig query evaluation algorithms [1, 2, 5, 9] have been proposed in the literature. Bruno et al. [2] designed the notable and efficient algorithm TwigStack. The algorithm pushes only nodes that are sure to contribute to the final results onto stacks for ancestor-descendent queries. Like most other algorithms, it is a two-phase algorithm, a path matches generation phase followed by a merge phase. Aiming to eliminate the two-phase overhead, Twig2Stack [11], a bottom-up evaluation algorithm, was proposed. Unfortunately, it may push nodes that do not contribute to the final results onto stacks, resulting in some extra work. It utilizes PathStack [2] to reduce runtime memory usage, however, at the price of increased stack manipulation complexity. TJFast [7] employs an innovative encoding scheme, called the extended Dewey code. Although this code can represent the relationships of nodes on a path elegantly, tremendous storage overhead is incurred, especially for longer paths. There are also some other algorithms [6, 4] that attempt to improve the performance of Twigstack for parent-child queries for which Twigstack is suboptimal.

## 3 A One-Phase Holistic Twig Join Algorithm

In this section, we present a one-phase twig query evaluation algorithm, called the HolisticTwigStack, based on the TwigStack. The new algorithm preserves all the strengths of the TwigStack and yet has no the aforementioned two-phase overhead. In the following, we illustrate the shortcomings of the TwigStack and Twig2STack by an example.

**Example.** Consider the data tree (a) and twig query (b) in Fig1.

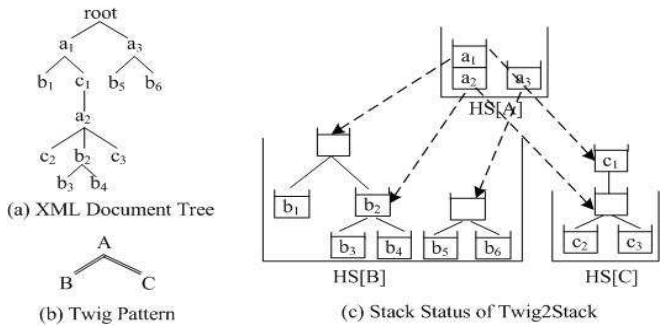


**Fig. 1.** Example of Holistic Twig Join Algorithm

TwigStack generates 12 path matches in the first phase: $a_1/b_1$, $a_1//b_2$, $a_1//b_3$, $a_1//b_4$, $a_2/b_2$, $a_2//b_3$, $a_2//b_4$ for A//B and $a_1/c_1, a_1//c_2, a_1//c_3, a_2/c_2, a_2/c_3$ for A//C. Several nodes, such as $a_1$ and $a_2$, appear multiple times in the matches, resulting in a large intermediate result (larger than the entire data tree itself). A 2-way merge is then needed to merge path matches into twig matches.

Twig2Stack is a one-phase algorithm. Although it does not have the two-phase overhead of TwigStack, it lacks the important advantage of TwigStack, i.e., not pushing any node that does not contribute to the twig matches onto a stack. In the example, Twig2Stack pushes non-contributing nodes $a_3$, $b_5$, and $b_6$, onto stacks. It also creates additional stacks, such as the one connecting $c_2$ and $c_3$, to speed up later query processing, at the cost of increased space complexity.

### 3.1 Notations

Like most twig query processing approaches [1, 2, 5, 9], we adopt the region code scheme. Each node in the XML document tree is assigned a unique 3-ary tuple: ($leftPos$, $rightPos$, $LevelNo$), which represents the left, right positions, and level number of the node, respectively. As in all the stack-based algorithms, there is a stream $T_q$ associated with each pattern node q of the twig query. The elements in each stream are sorted by their leftPos.

We define the $Top\_Branch\_Node$ as the branch node in the twig pattern at the highest level. The $Top\_Branch\_Node$ and the nodes above it in the twig pattern are called $Upper\_Nodes$. $Lower\_Nodes$ refer to nodes that are below the $Top\_Branch\_Node$ in the twig pattern. For example, the $Top\_Branch\_Node$ in Fig 1(b) is A. The document nodes that have the same type of $Top\_Branch\_Node$, $Upper\_Node$ and $Lower\_Nodes$ are referred as $Top\_Branch\_Element$, $Upper\_El$ -ement and $Lower\_Element$, respectively. We also define ClosestPatternAncestor(e) as the closest ancestor of the element e in the document tree according to the query pattern. For example, in Fig 1(a),ClosestPatternAncestor($b_2$)=ClosestPatternAncestor($b_3$)= ClosestPatternAncestor($b_4$)=$a_2$. We also define a pattern sibling element of e as an element in the document tree that (i) has the same node type as e; (ii) shares the same closest pattern ancestor with e, and (iii) does not have the ancestor-descendent relationship with e. In our example, $b_3$ is a pattern sibling element of $b_4$.

### 3.2 Stack Structure

The overheads incurred in the TwigStack algorithm are caused by the"hasty" output of the individual path matches generated when a leaf element is encountered, which splits the twig matches. We also observe that individual path matches are to be merged together based on the common branch nodes to form the twig matches. In order to keep the holisticity of the twigs, we opt to delay the output of individual path matches and their merging by holding them in lists of stacks until all elements under the same $Top\_Branch\_Element$ are processed. To hold multiple matching paths, we associate each $Upper\_Node$ with a single

stack and *Lower_Node* with lists of stacks, as compared with one stack for each query node in the TwigStack.

Like other stack-based algorithms, an element is pushed onto a stack whose top element is an ancestor of the incoming element. In addition, we require that the incoming element must have the same closest pattern ancestor as the top element. For an element that does not satisfy the above conditions, it will be stored in a new stack. Elements that have the same closest pattern ancestor will be linked together for convenient access. Fig 2 shows the stack structure of our algorithm, named HolisticTwigStack, of the given example in Fig 1. In the structure, we have represented the closest pattern ancestor relationships by solid arrows, e.g., $a_1$ to $S_{B1}$. The stacks of the same node type that share the same closest pattern ancestor are linked by dotted arrows. Take $S_{B2}$ and $S_{B3}$ as an example. They are linked together by using a dotted arrow pointing from $b_3$ to its sibling element $b_4$. Only 9 nodes are stored in HolisticTwigStack, which is less than Twig2Stack and TwigStack.

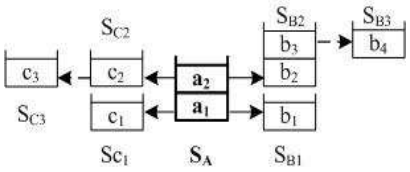


**Fig. 2.** Data Structure of HolisticTwigStack

It is observed that the commonly referenced ancestor-descendent relationships between query nodes, such as A//B and A//C, can be easily inferred from the closest pattern ancestor relationships by assuming that the ancestor elements of a stack "inherit" the relationships (i.e., the solid arrows) of their descendants (in the same stack). For example, $a_1$ could inherit the solid links of $a_3$, that is, $a_1$ implicitly also has links to $S_{B2}$ and $S_{C2}$ and thus the ancestor-descendent relationships (i.e., A//B, A//C) that $a_2$ has with other query nodes. The twig matches for the elements in the root stack can be constructed easily by traversing the links following the query pattern.

### 3.3 Algorithm

Our algorithm, named *HolisticTwigStack*, is presented in Algorithm 1. The algorithm computes the answer to a query twig pattern Q in one phase.

Elements are checked for satisfaction of structural relationships in the same way as is done in TwigStack by recursively calling the function *getNext()*, which is defined in TwigStack [2]. We attempt to withhold the elements sharing the same $Top\_Branch\_Element$ in memory until the element with the disjoint range arrives. The reason lies in the fact that the arrival of the disjoint element actually eliminate the possibility of the subtree rooted at this $Top\_Branch\_Element$ to

participate any further match. Like in the TwigStack, an incoming element e is pushed onto a stack (line 18) only if it is surely to contribute to a twig match (line 5); otherwise, we simply advance to next element (line 20). Before pushing e onto a stack, we further check if it falls beyond the range covered by the top element in the stack of $Top\_Branch\_Node$, namely, $S_{TBN}$ (line 6). Note that the top element covers only a subrange of other lower elements' ranges in the same stack, so the elements in $S_{TBN}$ are visited top-down until $S_{TBN}$ is empty or the current top element can cover the incoming element e . If e is disjoint with the current top element of the $S_{TBN}$ and all the paths under it have already been formed, twig matches sharing the same $Top\_Branch\_Element$ are output(line 8). Furthermore, the top element of $S_{TBN}$ is popped out and its ancestor element in $S_{TBN}$ (if there is any) inherit its relationship with its descendent elements in twig pattern(line 14). Otherwise, all the elements in the stacks are cleaned out (line 12). The same process also need to be conducted over the stacks of $Upper\_Node$ in order to clear all the nodes that are unnecessary for future matches(line 17). After having reached the end of streams (end(Q)), we output the remaining twig matches related to root elements left in $S\_root$ (line 22-30).

The basic idea of procedure *MoveElementToStack()* is given below. More details can be found in [10]. If the incoming element e is an $Upper\_Element$, we simply push it onto the corresponding stack. If e is a $Lower\_Element$, a more complicated process is involved. Assume the type of e is q and the stacks for the $Lower\_Node$ q have been numbered in their order of creation as $S_{q_1}, S_{q_2}, \ldots, S_{q_n}$. For each incoming element of type q , we check if it is a descendent of the top element of the last stack $S_{q_n}$ and if it has the same closest pattern ancestor as the top element of the stack. (The correctness of only checking the last stack will be given in the next subsection.) If so, we push the element onto the last stack $S_{q_n}$; otherwise, we push the element onto a new stack $S_{q_{n+1}}$. In the latter case, we shall check if it has the same closest pattern ancestor as any top element of a stack on an existing list. If so, we append $S_{q_{n+1}}$ to the end of that list; Otherwise, we directly link $S_{q_{n+1}}$ to its closest pattern ancestor.

*ShowTwigSolution()* is called to output twig matches rooted at the current root element. The twig matches can be formed by following the solid and dotted links between stacks and the ancestor-descendent relationships between the elements in the same stacks. Interested readers are referred to [10] for details of the algorithm.

### 3.4 Analysis of Algorithm

In this section, we show the correctness of the *HolisticTwigStack* algorithm. Due to the space limitation, readers are referred to [10] for details and proofs.

First, we introduce some terms and properties of TwigStack. subtreeNodes(q) include node q itself and all its descendants in the query pattern Q. An element has a minimal descendent extension if there is a solution for the sub-query rooted at q, composed entirely of the head elements for subtreeNodes(q). Here, the head element of q, denoted as $h_q$, is defined as the first element in $T_q$ that participates in a solution for the sub-query rooted at q [2].

TwigStack ensures that the element $e_q = next(T_q)$ is pushed onto the stack if and only if (i) element next($T_q$) has a descendent element $e_{q_i}$ in each of the stream $T_{q_i}$, for $q_i$=children(q), and (ii) each of the element $e_{q_i}$ recursively satisfies the first property [2].

**Algorithm 1**: HolisticTwigStack (Q)

```
1  begin
2      while not end(Q) do
3          q=getNext(Q);
4          e=the current first element in the stream of q;
5          if (q is of root type) or (at least one element in S_parent(q) covers e) then
6              while (!Empty(S_TBN) and e is disjoint with S_TBN's top element)
               do
7                  if (no leaf stack is empty) then
8                      ShowTwigSolution(S_root, S_root.size-1);
9                  end
10                 TopElem=s.pop();
11                 if Empty(S_TBN) then
12                     Clean all child stacks of s;
13                 else
14                     Update the links between stacks;
15                 end
16             end
17             Remove all the elements in each S_Upper_Node that are disjoint with e
               and update the links appropriately;
18             MoveElementToStack(q, e);
19         end
20         AdvanceList(q);
21     end
22     while not empty(S_root) do
23         ShowTwigSolution(S_root, S_root.size-1);
24         S_root.Pop();
25         if not Empty(S_root) then
26             Update the links between stacks;
27         else
28             Clean all child stacks;
29         end
30     end
31 end
```

**Lemma 1**. Let $e_1, e_2, \ldots, e_m$ be the sequence of elements pushed onto the stacks during the execution of the algorithm. Then, $e_1.left < e_2.left < \ldots < e_m.left$.

**Lemma 2**. The elements popped out of stack of *Upper_Node* at line 10, 17 24 and elements deleted at line 12 and line 28 from the stack lists of *Lower_Node* participate no further matches. So, the deletions are safe.

Earlier in the *MoveElementToStack()* procedure, when an element is to be pushed onto an appropriate stack, instead of examining all the stacks on the list

we only check if it is a descendent of the top element of the last stack (of its type) and if it has the same closest pattern ancestor as the top element. This optimization is based on the following lemma.

**Lemma 3:** For each incoming *Lower_Element*, either it can only be a descendent of the top element of the last stack (of its type), or it is not a descendent of any top element of the stacks (of its type).

**Lemma 4**: MoveElementToStack(q) correctly places the elements onto stacks and links elements to their closest pattern ancestors.

Thus the elements in the stacks can be reached in any case as long as it participates the final match, which is guaranteed by the property of getNext(). The relevant proof can be found in [2].

**Theorem 1**: Given a twig query pattern Q and an XML document tree D, Algorithm HolisticTwigStack correctly returns all answers to Q on D.

**Theorem 2**: Given a query twig pattern Q, comprising of n nodes and only ancestor-descendent edges, over an XML document D, Algorithm *HolisticTwigStack* has the worst-case I/O and CPU time complexities linear in the sum of sizes of the n input lists and the output list. Furthermore, the worst-case space complexity of *HolisticTwigStack* is the sum of the sizes of the n input lists.

Let us make simple comparisons with TwigStack. Our algorithm may take a little more CPU time in stack manipulation, but TwigStack would require extra time and space to store and merge the individual path matches. It is important to note that our stacks store twig matches rooted at elements that are currently in the stack of *Top_Branch_Node*, while TwigStack stores all the twig matches, in the form of individual path matches. Furthermore, nodes shared by multiple paths would have to be stored repeatedly in individual paths. Therefore, our algorithm in general uses much less space than TwigStack. For example, we are given a twig pattern where the root A has k children: A//$C_1$[//$C_2$]...[//$C_k$] and the document tree has n A nodes matching the given pattern. Each node $A_i$ (1≤i≤n) has $n_{i_1}$ $C_1$ children, $n_{i_2}$ $C_2$ children,...,$n_{i_k}$ $C_k$ children. Thus the total number of matches is $\sum_{i=1}^{n}\prod_{j=1}^{k} n_{ij}$. Both methods need the same time $O(\sum_{i=1}^{n}\prod_{j=1}^{k} n_{ij})$ to form all the combinations. But our method requires only $O(Max_{i,1\leq i\leq n}(\sum_{j=1}^{k} n_{ij})+1)$ space while TwigStack needs $O(\sum_{i=1}^{n}\sum_{j=1}^{k}(n_{ij}+1)) = O(\sum_{i=1}^{n}\sum_{j=1}^{k} n_{ij}+nk)$ to store the intermediate results. The space consumption of TwigStack can be further exacerbated when the twig pattern is deeper or more common nodes are shared by leaf nodes. In worst case, the intermediate result size of TwigStack is O(K×P)=O(J), where K is the sum of the lengths of the input lists for all leaf nodes, and P is the length of the longest root-to-leaf path in the twig pattern.

## 4 Experimental Results

In this section, we evaluate the performance of the *HolisticTwigStack* algorithm against TwigStack and Twig2Stack using both synthetic and real datasets.

Twig2Stack was shown to have better perfromance than TJFast [7] in [11], and thus the latter is ommitted here.

### 4.1 Experimental Set-up

The synthetic datasets consist of XMark, TreeBank, and other datasets generated by the random generator in [2]. The depths of the randomly generated trees vary from 5 to 10, fan-outs from 2 to 10, and the number of labels is set at 7. DBLP is the real dataset used in the experiments.

We use three types of twig queries in the experiments: Q1 represents a set of shallow but wide queries (dept=2, width= 4 to 5, in the patterns); Q2 a set of deep but narrow queries (dept= 3 to 5, width=2) and Q3 balanced queries (dept= 2 to 3 and width= 2 to 3).

We store the intermediate results of TwigStack in memory. This provision, eliminates the expensive output/input cost of the TwigStack, however, at the price of memory consumption.

### 4.2 Experimental Results

TwigStack's time is broken into two parts to reflect the cost of the two-phase algorithm with the lower and upper parts of the bars in the figures corresponding to the first and second phases, respectivley. Fig3 (a) and (b) show the performance on the random datasets. As observed, HolisticTwigStack is the fastest. TwigStack is slow for two reasons: (i) the recursive execution of ShowSolutionWithBlocking is time-consuming; and (ii) the scan and merge of the intermediate results in the second phase requires an extra amount of time. The entire execution times of TwigStack are 18%, 7% and 34% longer than ours for the three types of queries, respectively. Twig2Stack turns out to be the slowest. It requires nodes to be pushed/popped onto/out of stacks twice before forming the final results, incurring higher cost in stack manipulation. It takes 2.2, 1.9 and 1.3 times longer than HolisticTwigStack for the three types of queries, respectively. Fig3 (b) shows the space utilization. TwigStack uses, on average, 7,328, 5,208 and 9,192 bytes for Q1, Q2 and Q3, respectively, while ours uses only 1,068, 1,464 and 984 bytes for the same queries. This is because TwigStack stores all path matches in memory (due to our provision to reduce time consuming disk I/O), while ours stores only twig matches that are rooted at elements currently in the root stack, a subset of theirs, in memory. As for the Twig2Stack, although it does not store intermediate results, it may push nodes that do not contribute to the final results onto stacks, referred to as non-optimality. Recall that TwigStack and our method (based on the TwigStack), guarantees such optimality. It uses more space, from 9% to 209%, than our method, though better than TwigStack.

Figures 3(c) and(d) show the results of XMark. TwigStack consumes 8% to 18% more time than ours, as shown in (c) and 4 to 5,186 times more space than ours, as shown in (d). Please note in some cases, the space of HolisticTwigStack and Twig2Stack is too small to be shown. Usually, the larger the number of matches, the larger the space usage ratio. Our method is also much faster than

Twig2Stack; it consumes only 15%, 96% and 59% of Twig2Stack's time. However, the space utilizations are almost same. This is due to the uniform and balanced tree structure of XMark and the relatively uniform distribution of query matches.

Similar results are observed on TreeBank, which are shown in (e) and (f).

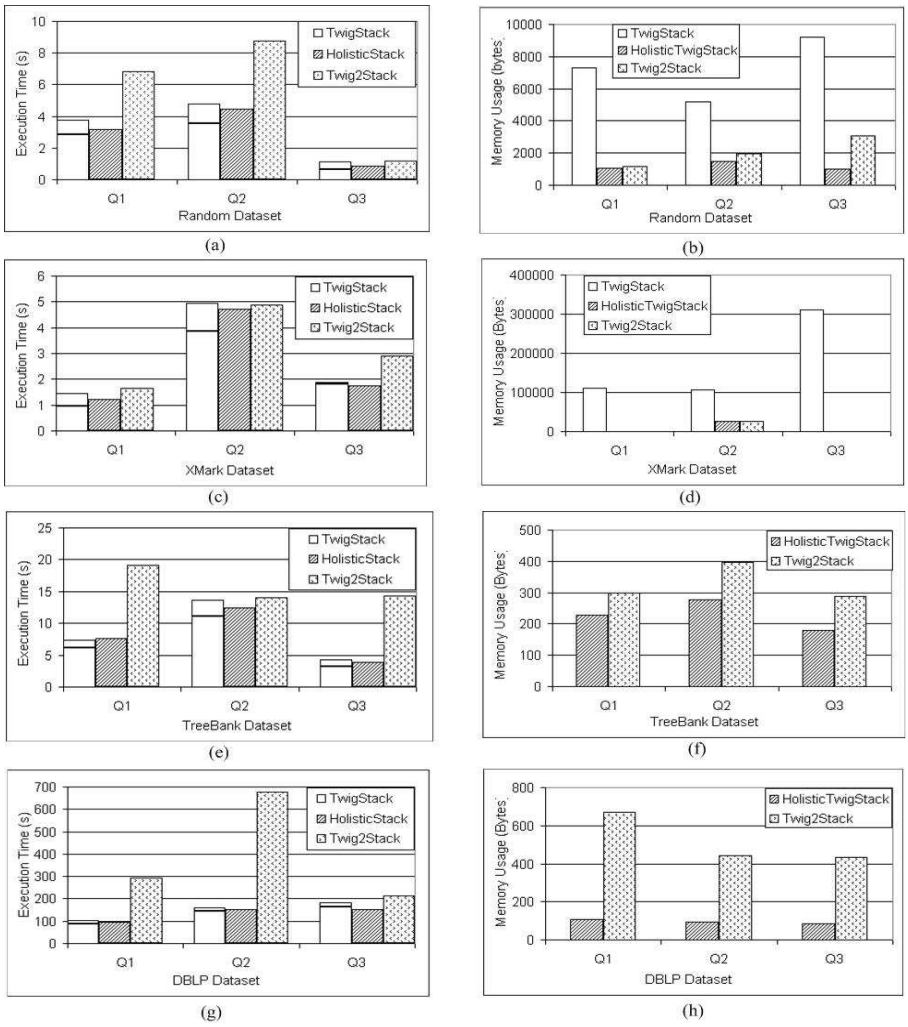


**Fig. 3.** Experimental Results

The results of DBLP are shown in (g) and (h). Although TwigStack is only slightly slower than ours, its intermediate result sizes are too large (around 100,000 times larger than the other two methods) to show in the chart. Twig2Stack uses much more time, 3.1, 4.4, and 1.4 times more, than our method, due to its

complex stack manipulation and overhead for processing "non-productive" nodes in the stacks (i.e., the non-optimality). Note that Twig2Stack uses much more space than ours in this experiments than in the XMark and TreeBank as there are more "non-productive" being pushed onto stacks.

In summary, our algorithm generally runs faster and requires less memory than TwigStack and Twig2Stack. The larger the query result sizes, the better our algorithm, compared with TwigStack. The more complex the tree structures, the better our algorithm, compared with Twig2Stack.

## 5 Conclusion

In this paper, we propose an efficient one-phase holistic twig pattern matching algorithm based on the TwigStack. We lower the expensive time and space overhead incurred in the two-phase algorithms by devising a novel stack structure to hold matching paths until entire twig matches that share the same *Top_Branch_Element* are formed. Experimental results have confirmed that our method is significantly more efficient in all cases tested.

## References


1. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proceedings of IEEE ICDE Conference (2002) 141–152
2. Bruno, N., Koudas, N., Srivastava, D.: Holistic Twig Joins: Optimal XML Pattern Matching. SIGMOD Conference (2002) 310–321
3. Chen, S., Li, H., Tatemura, J., Hsiung, W., Agrawal, D., Candan, K.: Twig2Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. SIGMOD Conference (2006) 283–294
4. Chen, T., Lu, J., Ling, T.W.: On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. ACM SIGMOD international conference (2005) 455–466
5. Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic Twig Joins on Indexe XML Documents. Proceedings of the 29th VLDB conference (2003) 310–321
6. Lu, J., Chen, T., Ling, T.W.: Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. Proceedings of CIKM (2004) 533–542
7. Lu, J., Ling, T.W., Chan, C-Y, Chen, T.: From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern. VLDB (2005) 193–204
8. Polyzotis, N., Garofalakis, M., Ioannidis, Y: Selectivity Estimation for XML Twigs. ICDE (2004) 264–275
9. Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman,G.:On Supporting Containment Queries in Relational Database Management Systems. SIGMOD (2001) 425–436
10. Jiang, Z., Luo, C., Hou, W., Zhu, Q., Wang, C-F: An Efficient One-Phase Holistic Twig Join Algorithm for XML Data. http://www.cs.siu.edu/∼zjiang (2005)
11. Chen, S., Li, H., Tatemura, J., Hsiung, W., Agrawal, D., Candan, K: Twig2Stack: Bottom-up Processing of Generalized-Tree-Pattern Queryies over XML Documents. VLDB(2006) 283–294