Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach

Jiaheng Lu, Ting Chen and Tok Wang Ling
School of Computing National University of Singapore
3 Science Drive 2, Singapore 117543
{lujiahen,chent,lingtw}@comp.nus.edu.sg

ABSTRACT

With the growing importance of semi-structure data in information exchange, much research has been done to provide an effective mechanism to match a twig query in an XML database. A number of algorithms have been proposed recently to process a twig query holistically. Those algorithms are quite efficient for quires with only ancestor-descendant edges. But for queries with mixed ancestor-descendant and parent-child edges, the previous approaches still may produce large intermediate results, even when the input and output size are more manageable. To overcome this limitation, in this paper, we propose a novel holistic twig join algorithm, namely *TwigStackList*. Our main technique is to look-ahead read some elements in input data steams and cache limited number of them to *lists* in the main memory. The number of elements in any list is bounded by the length of the longest path in the XML document. We show that *TwigStackList* is I/O optimal for queries with only ancestor-descendant relationships below branching nodes. Further, even when queries contain parent-child relationship below branching nodes, the set of intermediate results in *TwigStackList* is guaranteed to be a subset of that in previous algorithms. We complement our experimental results on a range of real and synthetic data to show the significant superiority of *TwigStackList* over previous algorithms for queries with *parent-child* relationships.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: [Systems-query processing]

General Terms

Algorithm, Performance

Keywords

XML, Holistic twig pattern matching

1. INTRODUCTION

XML is emerging as a *de facto* standard for information exchange over the Internet. Although XML documents could have rather complex internal structures, they can generally be modelled as *ordered* trees. In most XML query languages (see, e.g. [2, 3]), the structures of XML documents are expressed by *twig* (i.e. a small tree) patterns, while the values of XML elements are used as part of selection predicates. Finding all occurrences of a twig pattern in an XML database is a core operation in XML query processing, both in relational implementations of XML databases and in native XML databases [5]. In the past few years, many algorithms [5, 6, 9, 11, 12, 13, 14, 17] proposed in the literature are based on a form of labeling scheme that encodes each element in an XML database by its positional information. In order to answer a query twig pattern, these algorithms access the labels alone without traversing the original XML documents

In particular, Al-Khalifa et al. [1] propose to decompose the twig pattern into many binary relationships, and then use *Tree-merge* or *Stack-tree* algorithms to match the binary relationships, and finally stitch together basic matches to get the final results. The main disadvantage of such a decomposition based approach is that intermediate result sizes can get very large, even when the input and the final result sizes are much more manageable. To address the problem, Bruno et al. [5] propose a holistic twig join algorithm, namely *TwigStack*. With a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths, their approach is I/O and CPU optimal among all sequential algorithms that read the entire input for twigs with only *ancestor-descendant* edges.

The work reported in this paper is motivated by the following observation: although *TwigStack* has been proved to be I/O optimal in terms of input and output sizes for queries with only *ancestor-descendant* edges, their algorithms still cannot control the size of intermediate results for queries with *parent-child* edges. To get the better understanding of this limitation, we experimented with *TreeBank* dataset which was downloaded from University of Washington XML repository [16]. We use three twig queries patterns(as shown in Table 1), each of which contains at least one *parent-child* edge. *TwigStack* operates two steps:(i) a list of intermediate path solutions is output as intermediate results;(ii) the intermediate path solutions in the first step are merge-joined to produce the final solutions. Table 1 shows the numbers of intermediate path solutions output in the first step and the merge-joinable paths among them in the second step. An immediate observation from the table is that *TwigStack* output too many partial paths that are not merge-joinable. For all three queries, more than 95% partial paths produced by *TwigStack* in the first step are "useless" to final answers. Thus, our experiment shows that it is a big challenge to improve the previous algorithms to answer queries with *parent-child* edges.

Query	Partial paths	Merge-joinable paths	Percentage of useless paths
VP[/DT]//PRP_DOLLAR_	10663	5	99.9%
S[/JJ]/NP	70988	10	99.9%
S[//VP/IN]//NP	702391	22565	96.8%

Table 1: Number of partial path solutions produced by *TwigStack* against TreeBank data

In this paper, we propose a new holistic twig join algorithm, which has the same performance as *TwigStack* for query patterns with only ancestor-descendant edges, but it is significantly more efficient than *TwigStack* for queries with the presence of parent-child edges. In particular, we propose Algorithm *TwigStackList* to match query twig patterns. The main technique of *TwigStackList* is to make use of two data structures: stack and list for each node in query twigs. A chain of linked stacks is used to compactly represent partial results of individual query root-leaf paths. We look-ahead read some elements in input data streams and cache limited number of them in the list. The number of elements in any list is bounded by the length of the longest path in the XML document. The elements in lists help us to determine whether an element possibly contributes to final answers.

Our contribution can be summarized as follows:

- We propose a novel holistic twig join algorithm, namely *TwigStackList*. When all edges below branching nodes in the query pattern are ancestor-descendant relationships, the I/O cost of our algorithm is only proportional to the sum of sizes of the input and the final output. In other words, unlike previous algorithms, our algorithm can guarantee the I/O optimality even for queries with parent-child relationships below non-branching nodes. This improved result mainly owe to the *look-ahead* technique of our algorithm.
- Furthermore, even when there exist parent-child relationships below branching nodes, we show that the intermediate solutions output by *TwigStackList* are guaranteed to be a subset of that by the previous algorithms.
- We present experimental results on a range of real and synthetic data, and query twig patterns. Our experiments validate our analysis results and show the superiority of *TwigStackList* over previous algorithms.

The rest of the paper proceeds as follows. We first discuss the previous algorithm and show our intuitive observation in Section 2. The novel algorithm *TwigStackList* is presented in Section 3. We report the experimental results in Section 4. Section 5 is dedicated to the related work and Section 6 concludes this paper.

2. BACKGROUND

2.1 Data model and twig pattern query

XML data is commonly modelled by a tree structure, where nodes represent elements, attributes and texts, and parent-child edges represent element-subelement, element-attribute and element-text pairs. Most existing XML query processing algorithms [5, 9, 14] use a region code (*start, end, level*) to present the position of a tree node in the data tree. The region encodings support efficient evaluation of structural relationships. Formally, element u is an ancestor of element v if and only if $u.start < v.start < u.end$. For parent-child relationship, we also check whether $u.level = v.level$ -1.

Queries in XML query languages make use of twig patterns to match relevant portions of data in an XML database. Twig pattern nodes may be elements, attributes and texts. Twig pattern edges are either parent-child relationships (denoted by "/") or ancestor-descendant relationships (denoted by "//"). If the number of children of a node is greater than one, then we call this node a *branching* node. Otherwise, when the node has only one child, it is a *non-branching* node.

Given a twig pattern T and an XML database D, a match of T in D is identified by a mapping from nodes in T to elements in D, such that: (i) 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. The answer to query T with m nodes can be represented as a list of *m-ary* tuples, where each tuple (t_1,...,t_m) consists of the database elements that identify a distinct match of T in D.

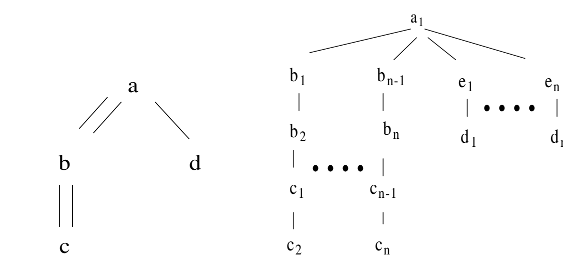
2.2 TwigStack and our observation

Bruno et al. [5] propose a novel holistic twig join algorithm called *TwigStack* to match XML twig patterns. They use a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths. In particular, *TwigStack* operates two phases as follows.

1. **Output path solutions** A list of root-leaf path solutions is output as intermediate path solutions. Each root-leaf solution matches the corresponding path pattern in the query pattern.
2. **Merge** All lists of path solutions in the first phase are merged to produce the final answer to the whole query twig pattern

When all edges in query patterns are ancestor-descendant (A-D) relationships, *TwigStack* ensures that each root-leaf solution in the first phase is merge-joinable with at least one solution to each of the other root-leaf query paths. Thus, none of those path solutions is redundant. However, this property does not hold if there is a parent-child (P-C) edge in the query pattern.

To see an example, if we evaluate the twig pattern in Figure 1(a) on the XML document in Figure 1(b), *TwigStack* will push a_1 into the stack and output all root-leaf path solutions: (a_1,b_1,c_1), (a_1,b_1,c_2),...,(a_1,b_{n-1},c_n), (a_1,b_n,c_n), because they match path $a//b//c$. Notice that in this example, there is no match at all! But *TwigStack* output $4n$ "useless" intermediate path solutions. Since the size of intermediate path solutions has a great impact on the performance



(a) Query twig pattern (b) An XML tree

Figure 1: Illustration to the sub-optimality of *TwigStack*

of holistic twig joins algorithms, in this paper, we focus on shrinking the size of intermediate path solutions for queries with P-C relationships.

The main problem of *TwigStack* is that it only considers ancestor-descendant property between nodes in the first phase. The level information of nodes, on the other hand, is *not* sufficiently exploited. As an illustration, see the query and data in Figure 1(a) and (b) again. Since $< a, d >$ edge in the twig pattern is the parent-child relationship, node a_1 in the document contributes to the final answer only if a_1 has a child with name d. But *TwigStack* pushes a_1 into the stack only because a_1 has a descendant (not a child) with tag d. Thus, this algorithm outputs a large size of intermediate paths. However, our method pushes a_1 into the stack only if a_1 or its descendant(with tag a) has a child with name d. In the document of Fig 1 (b), although a_1 has many descendants with tag d, none of them has a *child* with tag d. Thus, our method does *not* push a_1 into the stack and thereby avoid outputting the "useless" intermediate path solutions.

In the following, we extend the intuition in the above example and propose a new holistic twig matching algorithm, which is able to produce much less intermediate path solutions than *TwigStack* for queries with *parent-child* relationships.

3. TWIG JOIN ALGORITHM

In this section, we present *TwigStackList*, a new efficient algorithm for finding all matches of a query twig pattern against an XML document. We start this section with introducing some notations and data structures which will be used by *TwigStackList*.

3.1 Notation and data structures

A query twig pattern can be represented with a tree. The self-explaining function $isRoot(n)$ and $isLeaf(n)$ examine whether a query node n is a root or a leaf node. The function $children(n)$ gets all child nodes of n, and $PCRchildren(n)$, $ADRchildren(n)$ returns child nodes which has the parent-child or ancestor-descendant relationship with n in the query twig pattern, respectively. That is, $PCRchildren(n) \bigcup ADRchildren(n) = children(n)$. In the rest of the paper, "node" refers to a tree node in the twig pattern (e.g. node n), while "element" refers to an element in the data set involved in a twig join (e.g. element e).

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

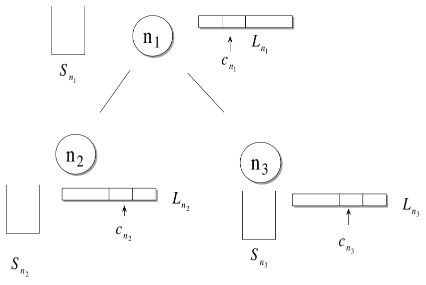


Figure 2: Stacks S_n and lists L_n used in our algorithm

Our join algorithm will make use of two types of data structure: list and stack. Given a query twig, we associate a list L_n and a stack S_n for each node n in the twig, as shown in Figure 2.

The use of stack in our algorithm is similar to that in *TwigStack*. That is, each data node in the stack consists of a pair: (positional representation of an element from T_n , pointer to an element in $S_{parent(n)}$). The operations over stack are: empty, pop, push, topStart, topEnd. The last two operations return the *start* and *end* attributes of the top element in the stack, respectively. At every point during computation: (i) the node in stack S_n (from bottom to top) are guaranteed to lie on a root-leaf path in the XML database (ii) the set of stacks contain a compact encoding of partial and total answers to the query twig pattern.

For each list L_n, we declare an integer variable say p_n, as a cursor to point to an element in the list L_n. We use $L_n.elementAt(p_n)$ to denote the element pointed by p_n. We can access the attribute values of $L_n.elementAt(p_n)$ by $L_n.elementAt(p_n).start$, $L_n.elementAt(p_n).end$ and $L_n.elementAt(p_n).level$. At *every* point during computation: elements in each list L_n are strictly nested from the first to the end, i.e. each element is an ancestor of the element following it. The operations over list L_n are delete(p_n) and append(e). The first operation delete $L_n.elementAt(p_n)$ in list L_n and the last operation appends element e at the end of L_n .

3.2 TwigStackList

Algorithm *TwigStackList*, which computes answers to a query twig pattern, is presented in Algorithm 2. This algorithm operates in two phases. In the first phase (line 1–11), it repeatedly calls the *getNext* algorithm with the query root as the parameter to get the next node for processing. We output solutions to individual query root-to-leaf paths in this phase. In the second phase (line 12), these solutions are merge-joined to compute the answer to the whole query twig pattern.

In Section 3.2.1, we explain the *getNext* algorithm and Section 3.2.2 presents the main algorithm in detail.

3.2.1 getNext algorithm

$getNext(n)$ is a procedure called in the main algorithm of *TwigStackList*. It returns a node n' (possibly $n' = n$) with three properties : assume that element $e_{n'}$=$getElement(n')$,

Algorithm 1 getNext(n)

```
1: if $isLeaf(n)$    return $n$
2: for all node $n_i$ in children($n$) do
3:     $g_i = getNext(n_i)$
4:      if ($g_i \neq n_i$)    return $g_i$
5: end for
6: $n_{max} = maxarg_{n_i \in children(n)}\ getStart(n_i)$
7: $n_{min} = minarg_{n_i \in children(n)}\ getStart(n_i)$
8: while ( $getEnd(n) < getStart(n_{max})$)    proceed($n$)
9: if ( $getStart(n) > getStart(n_{min})$)    return $n_{min}$
10: MoveStreamToList($n$, $n_{max}$)
11: for all node $n_i$ in PCRchildren(n) do
12:     if (there is an element $e_i$ in list $L_n$ such that $e_i$ is the
        parent of $getElement(n_i)$ ) then
13:        if ($n_i$ is the only child of $n$) then
14:           move the cursor $p_n$ of list $L_n$ to point to $e_i$
15:        end if
16:     else
17:        return $n_i$
18:     end if
19: end for
20: return $n$
```

Procedure $getElement(n)$

```
1: if $\neg empty(L_n)$  then
2:     return $L_n.elementAt(p_n)$
3: else return $C_n$
```

Procedure $getStart(n)$

```
1: return the start attribute of getElement(n)
```

Procedure $getEnd(n)$

```
1: return the end attribute of getElement(n)
```

Procedure $MoveStreamToList(n, g)$

```
1: while $C_n.start < getStart(g)$ do
2:     if $C_n.end > getEnd(g)$ then
3:        $L_n.append(C_n)$
4:     end if
5:     $advance(T_n)$
6: end while
```

Procedure $proceed(n)$

```
1: if $empty(L_n)$ then
2:     $advance(T_n)$
3: else
4:     $L_n.delete(p_n)$
5:     $p_n = 0$ {Move $p_n$ to point to the beginning of $L_n$}
6: end if
```

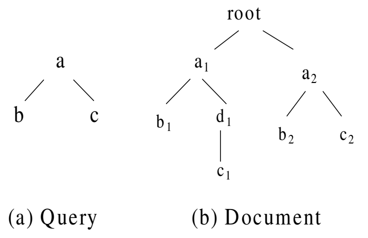


Figure 3: An example to illustrate *getNext* algorithm

then (i) $e_{n'}$ has a descendant e_{n_i} in each of stream T_{n_i} for $n_i \in children(n')$; and (ii) if n' is not a branching node in the query, element $e_{n'}$ has a child e_{n_i} in T_{n_i}, where $n_i \in PCRchildren(n')$ (if any); and (iii) if n' is a branching node, there is an element e_{n_i} in each T_{n_i} such that there exists an element e_i(with tag n) in the path from $e_{n'}$ to $e_{n_{max}}$ that is the parent of e_{n_i}, where $n_i \in PCRchildren(n')$ (if any) and $e_{n_{max}}$ has the maximal *start* attribute for all $children(n')$.

At line 2-5, in Algorithm *getNext*, we recursively invoke *getNext* for each $n_i \in children(n)$. If any returned node g_i is not equal to n_i , we immediately return g_i (line 4). Otherwise, we will try to locate a child of n which satisfies the above three properties. Line 6 and 7 get the *max* and *min* elements for the current head elements in lists or streams, respectively. Line 8 skips elements that do not contribute to results. If no common ancestor for all C_{n_i} is found, line 9 returns the child node with the smallest start value, i.e. g_{min} .

Line 10 is an important step. Here we look-ahead read some elements in the stream T_n and cache elements that are ancestors of $C_{n_{max}}$ into the list L_n. Whenever any element n_i cannot find its parent in list L_n for $n_i \in children(n)$, algorithm *getNext* returns node n_i (in line 17). Note that this step manifests the key difference between *TwigStackList* and the previous algorithms (i.e. *TwigStack*) . In this scenario, the previous ones return n instead of n_i , which may results in many "useless" intermediate paths. But our algorithm adopt a clever strategy: return n_i that has no parent in list L_n, since we make sure that n_i does not contribute to final results involved with the elements in the remaining parts of streams. Finally, if n is not a branching node, in line 14, we need to move the cursor in the list L_n to point to the parent of $getElement(n_i)$.

The main difference between two *getNext* algorithms in *TwigStack* and *TwigStackList* can be summarized as follows. In *TwigStack*, $getNext(n)$ return n' if the head element $e_{n'}$ in stream $T_{n'}$ has a descendant e_{n_i} in each stream T_{n_i}, for $n_i \in children(n')$ (which is the same as the first property as mentioned above), but *TwigStackList* needs n' to satisfy three properties, as illustrated as follows.

EXAMPLE 1. Consider a query twig pattern $a[/b]/c$ on a data set visualized in Figure 3. A subscript is added to each element in the order of their start values for easy reference. Initially, the three elements are (a_1,b_1,c_1). The first call of $getNext(root)$ returns node c, because element c_1 cannot find parent with tag a in the path from a_1 to it. But in this scenario, the first call of $getNext(root)$ of *TwigStack* would return a_1, since a_1 has two descendants b_1 and c_1 in stream b and c respectively. Because *TwigStack* return a_1 instead of c_1, in the main algorithm, *TwigStack* will output the useless path solution $< a_1, b_1 >$. Further, the second call of $getNext(root)$ in *TwigStackList* returns b_1. In addition, the cursor of node a is forwarded to a_2. Right before the third call, *TwigStackList* reach a cursor setup (a_2, b_2, c_2), which is actually the match of the query.

3.2.2 Main Algorithm

Algorithm 2 shows the main algorithm of *TwigStackList*. It repeatedly calls $getNext(root)$ to get the next node n to process, as described next.

First of all, line 2 calls *getNext* algorithm to identify the

Algorithm 2 TwigStackList

```
 1: while $\neg end()$ do
 2:   $n_{act} = getNext(root)$
 3:   if ($\neg isRoot(n_{act})$) then
 4:     cleanParentStack($n_{act}$,getStart($n_{act}$))
 5:   end if
 6:   if (isRoot($n_{act}$) $\vee\neg$empty($S_{parent(n_{act})}$)) then
 7:     clearSelfStack($n_{act}$,getEnd($n_{act}$))
 8:     moveToStack ($n_{act}$,$S_{n_{act}}$,pointertotop($S_{parent(n_{act})}$))
 9:     if (isLeaf($n_{act}$)) then
10:       showSolutionsWithBlocking($S_{n_{act}}$,1)
11:       pop($S_{n_{act}}$)
12:     end if
13:   else
14:     proceed($n_{act}$)
15:   end if
16: end while
17: mergeAllPathSolutions()
```

Function *end*()

```
1: return $\forall n_i \in subtreeNodes(n) : isLeaf(n_i) \bigwedge end(C_{n_i})$
```

Procedure $moveToStack(n, S_n, p)$

```
1: push (getElement(n),p) to stack $S_n$
2: proceed(n)
```

Procedure $clearParentStack(n, actStart)$

```
1: while ($\neg empty(S_{parent(n)}) \bigwedge$
        $(topEnd(S_{parent(n)}) < actStart)$) do
2:   pop($S_{parent(n)}$)
3: end while
```

Procedure $clearSelfStack(n, actEnd)$

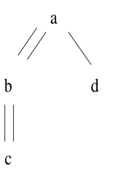
```
1: while ($\neg empty(S_n) \bigwedge (topEnd(S_n) < actEnd)$) do
2:   pop($S_n$)
3: end while
```

node n_{act} to be processed. Line 4 and 7 remove partial answers from the stacks of $parent(n_{act})$ and n_{act} that cannot be extended to total answer. If n is not a leaf node, we push element $getElement(n_{act})$ into $S_{n_{act}}$ (line 8); otherwise (line 10), all path solution involving $getElement(n_{act})$ can be output. Note that path solutions should be output in root-leaf order so that they can be easily merged together to form final twig matches (line 17). As a result, we may need to *block* some path solutions during output. Interested readers may refer to *TwigStack* [4] to know more details about *blocking* technique.

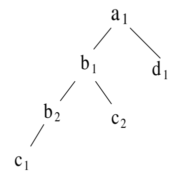
It is not correct to merge cleanParentStack and cleanSelfStack into one procedure cleanStack as Figure 4. Consider a twig query $a[//b//c]/d$ and a document in Figure 5. Suppose the four elements are initially at (a_1,b_1,c_1,d_1). At the first call of *getNext*, node a is returned. At this point, note that the current element pointed by cursor p_b in list L_b is b_2, instead of b_1 (recall, line 14 in Algorithm 1). Then the next two calls of *getNext* return b,c once to consume b_1 and c_1. After that, the current elements are (b_1,c_2,d_1) (stream a has finished). The next call of *getNext* will return node b again. Here, if we used the algorithm shown in Figure 4, then b_2 would not be popped from stack. Then, the property of stack (i.e. the upper element should be the descendant of the lower one) would not be hold.

```
......
3: if ($\neg$isRoot($n_{act}$) then
4:       cleanStack(parent($n_{act}$,getStart($n_{act}$))
5: endif
6: if (isRoot($n_{act}$) $\vee\neg$ empty($S_{parent(n_{act})}$) then
7:       cleanStack($n_{act}$,getStart($n_{act}$))
......
Procedure cleanStack(n,actStart)
1: while ($\neg$empty($S_n$)$\bigwedge$(topEnd($S_n$)<actStart)) do
2:       pop($S_n$)
```

Figure 4: The incorrect merge of two procedures



(a) Query twig pattern



(b) An XML tree

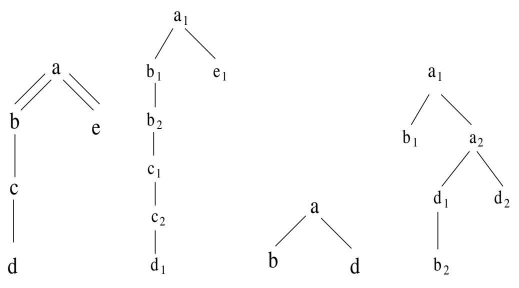
Figure 5: An example to show the incorrectness of codes in Fig. 4

Compared to the previous algorithm *TwigStack*, the benefit of the new algorithm *TwigStackList* can be illustrated with the following two examples.

EXAMPLE 2. Consider the query twig pattern Q_1 in Fig 6(a) and Doc_1 in Fig 6 (b). Initially, the first call of *getNext*() in *TwigStack* returns node a, but the first call of *getNext*() in *TwigStackList* returns node c. As a result, unlike algorithm *TwigStack*, *TwigStackList* does not output the intermediate result (a_1, e_1), which does not contribute to any final answers. □

EXAMPLE 3. Consider Q_2 in Fig 6(c) and Doc_2 in Fig 6 (d). Initially, the first call of *getNext* in *TwigStack* and *TwigStackList* return node a. After the second call of *getNext*, (a_1,b_1) is output as intermediate results in both *TwigStack* and *TwigStackList*. Subsequently, the third call of *getNext* in *TwigStack* returns node a again. But the third call of *getNext* in *TwigStackList* returns node b, since b_2 has not the parent in the stream T_a . Thus, unlike *TwigStack*, *TwigStackList* does not output the intermediate result (a_2,d_1) and (a_2,d_2), which do not contribute to any final answers. □

Example 2 illustrates the fact that, in *TwigStackList*, when twig patterns contain only ancestor-descendant relationships below branching nodes, each solution to individual query root-leaf path is guaranteed to be merge-joinable with at least one solution to each of the other root-leaf paths. On the other hand, Example 3 illustrates another fact that even if there exist parent-child relationships below branching nodes, *TwigStackList* is still superior to *TwigStack* in that it output less useless intermediate solutions.



(a) Q_1 (b) Doc_1 (c) Q_2 (d) Doc_2

Figure 6: Two examples to illustrate the benefits of Algorithm *TwigStackList*

3.3 Analysis of TwigStackList

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

DEFINITION 1. (*head element e_n*) In *TwigStackList*, for each node n in the query twig pattern, if the 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.

DEFINITION 2. (*child and descendant Extension*) We say that a node n has the child and descendant extension if the following three properties hold:

1. for each $n_i \in ADRchildren(n)$, there is an element e_{n_i} (with tag n_i) which is a descendant of e_n, and;
2. for each $n_i \in PCRchildren(n)$, there is an element e_i (with tag n) in the path from e_n to $e_{n_{max}}$ such that e_i is the parent of e_{n_i} , where $e_{n_{max}}$ has the maximal start attribute value for all head elements of child nodes of n; and
3. each of children of n has the child and descendant extension.

The above two definitions are important to establish the correctness of the following lemma.

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

- n' has the child and descendant extension.
- Either (a) $n = n'$ or (b) $parent(n)$ does not have the child and descendant extension because of n' (and possibly a descendant of n').

Using Lemma 1, we can prove the following lemma.

LEMMA 2. Suppose $getNext(n) = n'$ returns a query node n' ($n' \neq n$) in the line 17 of Algorithm *getNext*. If the stack is empty, then the head element does not contribute to any final solutions.

PROOF(SKETCH): Suppose that on the contrary, there is a solution using the head element. In line 10 of algorithm *getNext*, we insert all elements with the name $parent(n')$ which are in the path from $e_{parent(n')}$ to $e_{n_{max}}$ into the list $L_{parent(n')}$. According to line 12, if the parent of $e_{n'}$ is not in $L_{parent(n')}$, then using our hypothesis, we know that $parent(e_{n'})$ also participate in the final solution. But using Lemma 1, we see that this is a contradiction, since the start attribute of $parent(e_{n'})$ is less than that of $e_{parent(n')}$ and the stack $S_{parent(n')}$ is empty. □

LEMMA 3. At every point during computation of Algorithm *TwigStackList*: elements in each stack S_n are strictly nested, i.e. each element is a descendant of the element below it.

PROOF: This lemma is obvious in the previous *TwigStack*. But since algorithm *TwigStackList* may change the cursor of the list, this lemma is nontrivial. In *TwigStackList*, we can insert elements into the stack only in Procedure *moveToStack*. There are four cases for relationship between the new element e_{new} to be pushed into stack and the existing top element e_{top} in stack(see Figure 7).
Case(i): Since $e_{top}.end < e_{new}.end$, the element e_{top} will be popped in Procedure *cleanSelfStack* . So this case is impossible.
Case(ii): In this case, e_{new} will be added into the stack safely.
Case(iii): Similar to case (i), since $e_{top}.end < e_{new}.end$, the element e_{top} will be popped. We also ensure that e_{top} cannot participate in final answers any longer.
Case(iv): This case is impossible. Because, in algorithm *TwigStackList* , we can change the cursor of a list only in line 14 of *getNext*. The new element indicated by the cursor is guaranteed to be a descendant of the previous one.
Therefore, this lemma holds in all cases. □

LEMMA 4. In *TwigStackList*, any element that is popped from the stack S_n does not participate in any new solution any more.

PROOF: Any element is popped from stack S_n in either Procedure *cleanParentStack* or *cleanSelfStack*. In the following, we prove the correctness of the lemma in these two cases respectively.

- In *cleanParentStack*, suppose on the contrary, there is a new solution involving the popped element e_{pop}. According to line 1 of *cleanParentStack*, $e_{pop}.end <$ *actStart*, where *actStart* is the start attribute of the head element of parent(n) (i.e. $e_{parent(n)}$). Using the containment property, e_{pop} cannot be contained by any element in the path from the root to $e_{parent(n)}$ and after $e_{parent(n)}$, which is a contradiction.
- In *cleanSelfStack*, using the containment property, we see that *cleanSelfStack* pops elements that are descendants of e_n , where e_n is the head element of node n. The popped element does not participate in new answers any more. This is because, at this point, n has only one child with parent-child relationship. Thus, the start value of any child of e_{pop} is less than that of the head element of node $children(n)$. Thus, there is no element that is a child of e_{pop} in the remaining portion of the stream $T_{child(n)}$. Therefore, e_{pop} does not participate in any new solutions. □

THEOREM 1. Given a query twig pattern q and an XML database D, Algorithm *TwigStackList* correctly returns all answers for q on D.

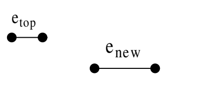
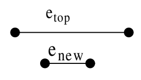
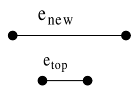
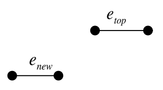
	Case(i)	Case(ii)	Case(iii)	Case(iv)
Property	e_{top}.end < e_{new}.start	e_{top}.start < e_{new}.start e_{top}.end>e_{new}.end	e_{top}.start > e_{new}.start e_{top}.end < e_{new}.end	e_{new}.end < e_{top}.start
Segment				

Figure 7: Illustration to the proof of Lemma 3

PROOF(SKETCH): We prove Theorem 1. Using Lemma 2, we know that when *getNext* returns a query node n in the line 17 of *getNext*, if the stack $S_{parent(n)}$ is empty, then 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 descendant and child extension is returned by *getNext* before e_n . By using Lemma 3 and Lemma 4, we can maintain, 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 solutions that use e_n. □

While correctness holds for query twig patterns with both ancestor-descendant and parent-child relationships in any edges, we can prove optimality only for the case where parent-child relationships appear only in edges below non-branching nodes. The intuition is that we push into stacks only elements that have the child and descendant extension. If there is a parent-child relationship below the non-branching node, according to Lemma 1, we are guaranteed that e_n is pushed into stack only if e_n has a child element in the stream $T_{child(n)}$. Therefore, we have the following result.

THEOREM 2. Consider a query twig pattern with m nodes, and there are only ancestor-descendant relationships below branching nodes (in other words, this pattern may have parent-child relationships below non-branching nodes), and an XML database D. Algorithm *TwigStackList* has the worst-case I/O complexities linear in the sum of sizes of the m input lists and the output list. □

Since the worst-case size of any stack and list in *TwigStackList* is proportional to the maximal length of a root-leaf path in the XML database, we have the following results about the space complexity of *TwigStackList*.

THEOREM 3. Consider a query twig pattern with m nodes and an XML database D. The worst-case space complexity of algorithm *TwigStackList* is proportional to m times the maximal length of a root-leaf path in D.

It is particularly important to note that, even for the case where query patterns contain parent-child relationships below branching nodes, as shown in our experimental results in Section 4, algorithm *TwigStackList* usually output much less intermediate path solutions than *TwigStack*. The reason is simple. *TwigStackList* pushes any element into stack that has both descendant and child extension, but *TwigStack* pushes any element that has only the descendant extension into the stack. Therefore, *TwigStackList* pushes fewer elements that do not contribute to final answers to the stack and thereby output less intermediate results.

4. EXPERIMENTAL EVALUATION

In this section we present experimental results on the performance of the twig pattern matching algorithms, namely *TwigStackList* and *TwigStack*, with both real and synthetic data. We evaluated the performance of these algorithms using the following metrics:

1. **Number of partial solutions.** This metric measures the total number of partial path solutions, which reflects the ability of the algorithms *TwigStackList* and *TwigStack* to control the size of intermediate results for different kinds of query twig patterns.

2. **Running time.** The running time of an algorithm is obtained by averaging the running times of several consecutive runs.

4.1 Experimental Setting

We implemented all algorithms in JDK 1.4. All our experiments were performed on 1.7GHz Pentium 4 processor with 768MB RAM running on windows XP system. We used the following three real-world and synthetic data sets for our experiments:

- **TreeBank.** We obtained the TreeBank data set from the University of Washington XML repository [16]. The document in the TreeBank is deep and has many recursions of element names. The data set has the maximal depth 36 and more than 2.4 million nodes.

- **DTD data set.** We used the following simple DTD to create highly and less nested data trees: $a \rightarrow bc|cb|d$; $c \rightarrow a$; where a and c are non-terminals and b and d are terminals. We generated about 114M bytes raw XML data according to this DTD. The maximal depth of each data tree varied from 3-30.

- **Random.** We generated random data trees using two parameters: *fan-out*, *depth*. The fan-out of nodes in data trees varied in the range of 2-100. The depths of data trees varied from 10-100. We use seven different labels, namely: a,b,c,d,e,f,g, to generate the data sets. The node labels in the trees were uniformly distributed.

4.2 TwigStackList Vs TwigStack

We compare algorithm *TwigStackList* against *TwigStack* with different twig pattern queries over above three data sets.

4.2.1 TreeBank

We first used the queries shown in Table 2 over the real-world Treebank data. These queries have different twig structures and different distribution of ancestor-descendant (A-D) and parent-child (P-C) edges. In particular, all edges in Q1 are A-D relationships, while all edges in Q2,Q6 are P-C relationships. In Q3,Q5, all edges below branching nodes are A-D relationships, but in Q4, edges below branching nodes contain both A-D and P-C relationships. We choose these different queries so that we can give a comprehensive comparison between Algorithm *TwigStackList* and *TwigStack*.

Query	XPath expression
Q1	S[//MD]//ADJ
Q2	S/VP/PP[/NP/VBN]/IN
Q3	S/VP//PP[//NP/VBN]//IN
Q4	VP[/DT]//PRP_DOLLAR_
Q5	S[//VP/IN]//NP
Q6	S[/JJ]/NP

Table 2: Queries over TreeBank data

Queries	*TwigStack* Path	*TwigStackList* Path	Reduction percentage	Useful Path
Q1	35	35	0%	35
Q2	2957	143	95%	92
Q3	25892	4612	82%	4612
Q4	10663	11	99.9%	5
Q5	702391	22565	96.8%	22565
Q6	70988	30	99.9%	10

Table 3: Number of intermediate path solutions produced by *TwigStack* and *TwigStackList* for TreeBank data

Figure 8(a) shows the execution time of queries for two algorithms and Table 3 shows the number of partial solutions, where the fourth column is the number of merge-joinable path that can contribute to at least one final answer. From the table and figure, we have several observations and conclusions:

- When the query twig pattern contains only ancestor-descendent edges, both *TwigStackList* and *TwigStack* are I/O optimal in that each of path solutions can contribute to final answers(see Q1 in Table 3). Thus, in this case, both algorithms have very similar performance(see Q1 in Fig 8(a)).

- When all edges below branching nodes contain only ancestor-descendant relationships, *TwigStackList* is still I/O optimal, but *TwigStack* has not the nice property. For example, see Q3 in Table 3. The numbers of intermediate path solution in *TwigStack* is 25892, while *TwigStackList* produces only 4612 solutions. Considering the number of merge-joinable path is also 4612, each of path solutions in *TwigStackList* contributes to final answers.

- When edges below branching nodes contain any parent-child relationship, both algorithms *TwigStackList* and *TwigStack* are suboptimal (see Q2,Q4,Q6 in Table 3). But in this case, we observed that the number of intermediate paths produced by *TwigStackList* is significantly less than that by *TwigStack*. For example, in queries Q4 and Q6, *TwigStack* produced 10663 and 70988 intermediate paths, while *TwigStackList* only produce 11 and 30 solutions. About 99% partial solutions of *TwigStack* are pruned by *TwigStackList*. Therefore, the execution time of *TwigStack* is considerably slower than that of *TwigStackList* for these queries.

In summary, Algorithm *TwigStackList* performs better than *TwigStack* for query twig patterns with parent-child edges.

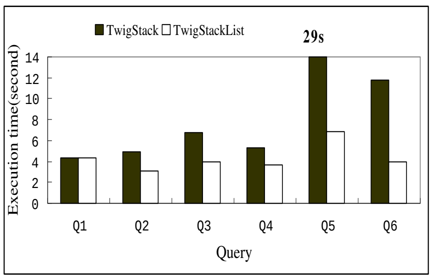
4.2.2 DTD data set

We then used the query $a[//c]//b/d$ over different synthetically generated data sets. Note that in this query, all edges below the branching node are ancestor-descendant relationships. According to the DTD rules $a \rightarrow bc|cb|d$ and $c \rightarrow a$, since b is a terminal and has not any child nodes, clearly, there is no answer for this query in the data set. So any path solution does not contribute to final answers. We varied the size of tag d relative to the size of tag b and c as the child of tag a from 10% to 90% . We generated nine data sets and each of them has about 1 million nodes. Figure 8 (b-c) show the execution time of *TwigStack* and *TwigStackList* and the number of partial path solutions each algorithm produces. The consistent gap between *TwigStack* and *TwigStackList* results from the fact that the latter is I/O optimal for this query, but the former is not. As seen in Figure 8(c), the number of solutions produced by *TwigStack* is very large, but *TwigStackList* does not produce any partial solutions at all!

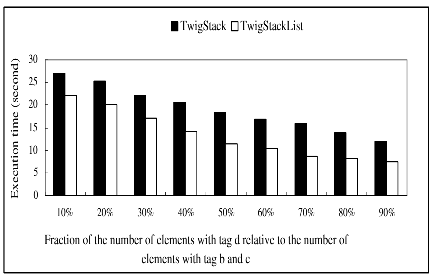
We issued the second Xpath query $a[/c][/d]/b$ over the previous nine data sets. As before, there is no match for the query in data sets. But the main different with the previous experiment is that *TwigStackList* is also not optimal in the second case (since there are parent-child relationships below the branching node a). Therefore, both *TwigStack* and *TwigStackList* output some intermediate path solutions that do not contribute to the final answers. Figure 8 (d) shows the execution time for two algorithms and Figure 8 (e) shows the number of partial solutions. We can see that even in the presence of parent-child relationship below the branch node , *TwigStackList* is again more efficient than *TwigStack*.

4.2.3 Random data set

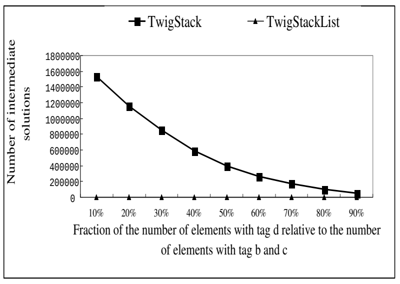
Finally we used random data set to compare *TwigStack* and *TwigStackList*. In particular, we generate random XML documents consisting of seven different labels, namely: a,b,c,d,e,f,g. The random data set has about 1 million nodes. We issued five twig queries shown in Figure 9, which have more complex twig structures than that of the queries in the previous experiments. The experimental results, including the execution time and the number of partial solutions are shown in Fig 8(f) and Table 4 respectively. From the figure and table, we see that for all queries, *TwigStackList* is again more efficient than *TwigStack*.



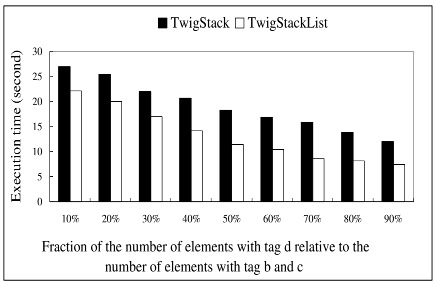
(a)Execution time on TreeBank



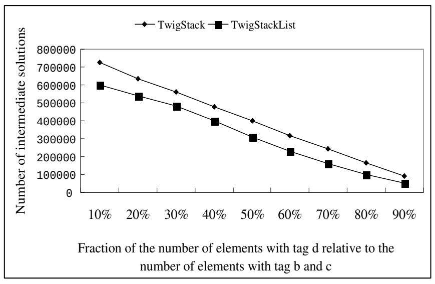
(b) Execution time for $a[//c]//b/d$



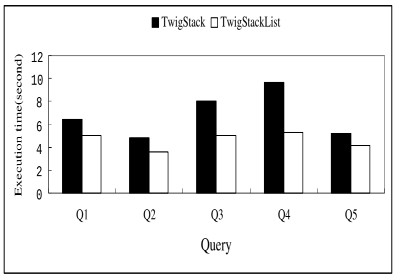
(c)Intermediate solutions for $a[//c]//b/d$



(d) Execution time for $a[/c][/d]/b$



(e) Intermediate solutions for $a[/c][/d]/b$



(f) Execution time on random data

Figure 8: (a) Execution time against TreeBank data (b)-(e) Performance comparison of two algorithms using generated DTD data (f) Execution time against random data

Queries	*TwigStack* Path	*TwigStackList* Path	Reduction percentage	Useful Path
Q1	9048	4354	52%	2077
Q2	1098	467	57%	100
Q3	25901	14476	44%	14476
Q4	32875	16775	49%	16775
Q5	3896	1320	66%	566

Table 4: Number of intermediate path soltuions produced by *TwigStack* and *TwigStackList* for random data

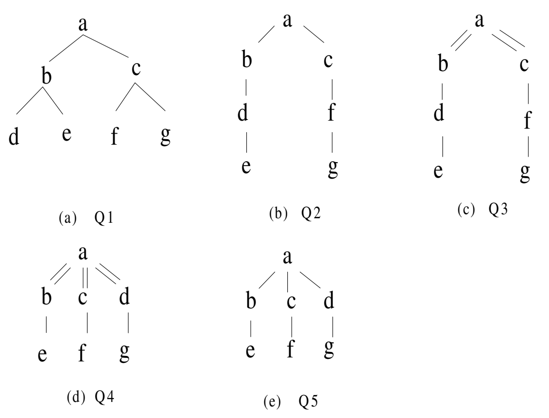


Figure 9: Queries against random data

Interestingly, if we compare Table 3 and 4, we find that most of the reduction percentage in Table 4 is smaller than that in Table 3. This fact is due to the difference between TreeBank and random data. There are more than 50 tags in the real-world TreeBank data, but there are only 7 tags in random data. Further, random data (average depth 50) is deeper than TreeBank (average depth 8). Therefore, for random data, in the line 12 of *getNext* algorithm, the IF condition usually returns true. But for TreeBank, the condition of line 12 usually returns false. Thus, more intermediate paths are pruned by *TwigStackList* in TreeBank than that in random data. From this fact, we conclude that (i) compared to *TwigStack*, *TwigStackList* can reduce partial path solutions for the queries that have parent-child relationship, and (ii) the reduction percentage is relative to the tag distribution in the data sets.

5. RELATED WORK

Join processing is central to query evaluation. In the context of semi-structured and XML databases, structural join is essential to XML query processing because XML queries usually impose certain structural relationships (e.g. parent-child or ancestor-descendant relationships). For binary structural join, Zhang et al [17]. proposed a multi-predicate merge join (MPMGJN) algorithm based on (*Start*, *End*, *Level*) labelling of XML elements. The later work by Al-Khalifa et al [1] gives a stack-based binary structural join algorithm. Then Wu et al [14] studied the problem of binary join order selection for complex queries on a cost model which takes into consideration factors such as selectivity and intermediate results size.

More recently, Bruno et al [5] propose a holistic twig join algorithm, namely *TwigStack*, to avoid producing a large intermediate result. *TwigStack* is I/O optimal for queries with only ancestor-descendant relationships. Jiang et al [9] studied the problem of holistic twig joins on all/partly indexed XML documents. Their proposed algorithms use indexes to efficiently skip the elements that do not contribute to final answers, but their method cannot reduce the size of intermediate results. Choi. et al [7] proves that optimality evaluation of twig patterns with arbitrarily mixed ancestor-descendant and parent-child edges is not feasible. In this paper, we proposed the algorithm *TwigStackList*, which is better than any of previous work in term of the size of intermediate results for matching XML twig pattern with both parent-child and ancestor-descendant edges.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose an enhanced holistic twig pattern matching algorithm *TwigStackList*. Unlike the previous Algorithm *TwigStack*, our approach takes into account the level information of elements and consequently results in much smaller intermediate path solutions for query twig patterns with both ancestor-descendant and parent-child edges. Experimental results showed that our method is much more efficient than *TwigStack* for queries with parent-child edges.

Since Choi et al.[7] have proven that there is no algorithm which is I/O and CPU optimal for all query patterns, one issue to improve our algorithm is to modify the coding method of the intermediate path solutions so that its size is guaranteed to be no more than the size of input data for all queries. Another possible issue involves designing a new powerful numbering scheme, which needs to change the format of input data.

7. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel. Y. Wu, N. Koudas, D. Srivastava "Structural Joins: A primitive for efficient XML query pattern matching" In Proceedings of ICDE 2002 pages 141-152

[2] A. Berglund , S. Boag , D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, J. Simeon "XML Path Language (XPath) 2.0" W3C Working Draft 22 August 2003

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu J. Robie , J. Simeon "Xquery 1.0: An XML QueryW3C" Working Draft 22 August 2003

[4] N. Bruno, N. Koudas, and D. Srivastava. "Holistic twig joins: Optimal XML pattern matching" Technical Report Columbia University March 2002

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

[6] Y. Chen, S. B. Davidson, Y. Zheng "BLAS: An Efficient XPath Processing System" In Proceedings of SIGMOD 2004, pages 47-58

[7] B. Choi, M. Mahoui, D. Wood "On the Optimality of Holistic Algorithms for Twig Queries" DEXA 2003 pages 28-37

[8] J.Hellerstein, J. Naughton, and A. Pfeifer "Generalized search trees for database systems" In Proceedings of VLDB, 1995 pages 562-573

[9] H. Jiang, W. Wang, H. Lu and J.X. Yu "Holistic twig joins on indexed XML documents" In Proceedings of VLDB 2003 pages 273-284

[10] H. Jiang, H. Lu, W. Wang, B. C. Ooi "XR-Tree: Indexing XML Data for Efficient Structural Joins" In Proceedings of ICDE 2003, pages 253-263

[11] H. Jiang, H. Lu, W. Wang "Efficient Processing of Twig Queries with OR-Predicates" In Proceedings of SIGMOD 2004, pages 59-70

[12] Q. Li and B. Moon "Indexing and querying XML data for regular path expressions" In Proceedings of VLDB 2001 pages 361-370

[13] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang "Storing and Querying Ordered XML Using a Relational Database System" In Proceedings of ACM SIGMOD 2002 pages 204-215

[14] Y. Wu, J. M. Patel, H. V. Jagadish "Structural Join Order Selection for XML Query Optimization" ICDE 2003 pages 443-454

[15] XML-benchmark http://monetdb.cwi.nl/xml

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

[17] C. Zhang, J.F. Naughton, D.J. Dewitt, Q. Luo and G.M. Lohman "On Supporting containment Queries in Relational Database Management Systems" In Proceedings of. ACM SIGMOD, 2001 pages 425-436