

TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-predicates on XML Data

Tian Yu, Tok Wang Ling, Jiaheng Lu

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

Abstract. As business and enterprises generate and exchange XML data more often, there is an increasing need for searching and querying XML data. A lot of researches have been done to match XML twig queries. However, as far as we know, very little work has examined the efficient processing of XML twig queries with not-predicates. In this paper, we propose a novel holistic twig join algorithm, called *TwigStackList¬*, which is designed for efficient matching an XML twig pattern with negation. We show that *TwigStackList¬* can identify a large query class to guarantee the I/O optimality. Finally, we run extensive experiments that validate our algorithm and show the efficiency and effectiveness of *TwigStackList¬*.

1 Introduction

In the recent years, business and enterprises generate and exchange XML data more often. The XML data can be very complex and deeply nested. Therefore, there is a lot of interest in query processing over data that conforms to a tree-structured data model ([2, 7]). Efficiently matching all twig patterns in an XML database is a major concern of XML query processing. Among them, holistic twig join approach has been taken as an efficient way to match twig pattern since it has shown effectiveness by reducing the intermediate result ([2–5]). We observe that, the existing work on holistic twig query matching only consider twig queries without not-predicate, such as:

Q1: *suppliersDatabase/supplier[./store]//part*

This twig pattern is written in XPath [15] format. It selects *part* elements which are descendants of *supplier* elements having at least one descendent element *store*.

However, in real applications, XML queries is more complex and may contain logical-NOT predicates (or not-predicates), such as:

Q2: *suppliersDatabase/supplier[NOT(./store)]//part*

The query selects *part* elements which are descendent of *supplier* elements having no descendant element *store*. Therefore, it is important for us to specify an algorithm to efficiently solve the twig patterns with not-predicates.

In general, the not-predicates can be used in a nested manner, such as:

Q3: *suppliersDatabase/supplier[NOT(./store[NOT(location = "Singapore")])]/part*

It selects *part* elements which are descendent of *supplier* elements having no descendent element *store* that is not in *Singapore*. In another word, if the supplier only contains *store* that is in “*Singapore*”, its descendent *part* is in the answer to query Q3.

We call the general twig queries with not-predicates as NOT-twig queries. The queries without not-predicates are called Normal-twig queries. The graphical representations of NOT-twig Q1, Q2, and Q3 are shown in Fig. 1(a), (b) and (c).

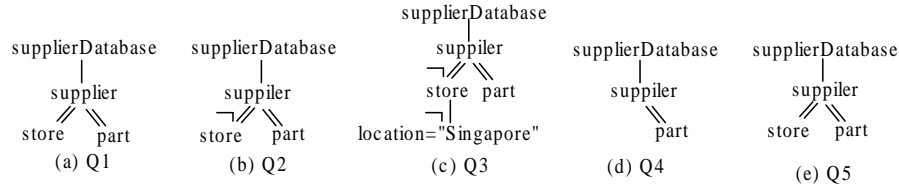


Fig. 1. Examples of XML Queries

To match a twig query with not-predicates, a naïve method is to decompose it into several Normal-twig queries (without not-predicates). Each decomposed Normal-twig queries are individually evaluated using the existing method, and the final result can be calculated based on the results of the individual decomposed quires. Each not-predicate in the NOT-twig produce an additional decomposed query.

For example, we can evaluate query Q2 by solving the following two queries:

Q4: *suppliersDatabase/supplier//part*

Q5: *suppliersDatabase/supplier[.//store]//part*

Existing holistic algorithms, *TwigStack* [2] or *TwigStackList* [7] can be used to find the answers of Q4 and Q5, shown in Fig. 1(d) and (e). The query Q2 can be evaluated by calculating the difference of two answering set for Q4 and Q5. Clearly, this naïve approach is not optimal in most cases. For example, the elements in *supplier* and *part* has to be accessed twice in order to evaluate the two decomposed queries from query Q2.

Jiao et al. [6] proposed a holistic path join algorithm for path query with not-predicates. However, it cannot answer the problem of twig pattern with not-predicates. To the best of our knowledge, this paper is the first that address the problem of XML NOT-twig matching.

In this paper, we developed a new algorithm to match NOT-twig queries holistically without decomposing them into Normal-twigs. The contributions of our work are:

- We discuss the problem of sub-query matching and propose a novel holistic twig join algorithm, namely *TwigStackList*[−], based on the new concept of *Negation Children Extension* (for short *NCE*). Unlike naïve method, this approach ensures that all elements in the XML documents are scanned no more than once.
- We demonstrate that in a NOT-twig, when all the positive edges below branching nodes are ancestor-descendant relationships, the I/O cost is only proportional to the sum of sizes of the input and the final output. Therefore, our algorithm can guarantee the I/O optimality for a very large query set. Furthermore, even when there exist positive parent-child relationships below branching nodes, the intermediate

solutions output by *TwigStackList*[¬] are guaranteed to be smaller than the naïve method.

- We present experimental results on a range of real and synthetic data, and query twig patterns. The results validate our analysis and show the superiority of *TwigStackList*[¬] in answering twig patterns with not-predicates.

The rest of the paper is organized as follows. Section 2 studies the related work. Section 3 defines the representation of twig queries with not-predicates and discusses the problem of sub-query matching. Section 4 explains our algorithm *TwigStackList*[¬], and proves its correctness. Section 5 presents the performance study and the experimental results. Finally, section 6 concludes the paper.

2 Related Work

With the increasing popularity of XML data representation, XML query processing and optimization has attracted a lot of research interest. In this section, we summarize the literature on matching twig queries efficiently.

Zhang et al. [16] proposed a multi-predicate merge join (MPMGJN) algorithm based on (DocId, Start, End, Level) labeling of XML elements. The Twig join algorithms by Al-Khalifa et al. [1] gave a stack-based binary structural join algorithm. The later work by Bruno et al. [2] proposed a holistic twig join algorithm, *TwigStack*, to avoid producing a large intermediate result. However, this algorithm is only optimal for ancestor-descendent edges. Therefore, Lu et al. [7] developed a new algorithm called *TwigStackList*, in which a list data structure is used to cache limited elements to identify a larger optimal query class. Chen et al. [3] studied the relationship between different data partition strategies and the optimal query classes for holistic twig join. Recently, Lu et al. [8] proposed a new labeling scheme called extended Dewey to efficiently process XML twig pattern.

In order to solve complex twig queries, Jiang et al. [4] researched the problem of efficient evaluation of twig queries with OR predicates. Lu et al. [9] studied how to answer an ordered twig pattern using region encoding. Jiao et al. [6] proposed a holistic path join algorithm for path query with not-predicates.

In the recent years, two new algorithms, *ViST* [13] and *PRIX* [10], are proposed to transform both XML data and queries into sequences, and answer XML queries through subsequence matching. Their methods avoid join operations in query processing. However, to eliminate false alarm and false dismissal, they resort to time consuming operations (post-processing for false alarm and multiple isomorphism queries processing for false dismissal [12]).

3 Preliminaries

3.1 XML Data Model

We model XML documents as ordered trees. The edges between the tree nodes can be parent-child (for short PC) or ancestor-descendant (for short AD).

Many state of the art join algorithms on XML documents are based on certain numbering schemes. For example, the binary XML structural join in [1, 16], and the twig join in [2] use $(startPos: endPos, LevelNum)$. It is an example of using *region encoding* to label elements in an XML file. Fig. 2 shows an example XML data tree. $startPos$ and $endPos$ are calculated by performing a pre-order (document order) traversal of the document tree; $startPos$ is the number in sequence assigned to an element when it is first encountered and $endPos$ is equal to one plus the $endPos$ of the last element visited. Leaf elements have same $startPos$ and $endPos$. $LevelNum$ is the level of a certain element in its data tree.

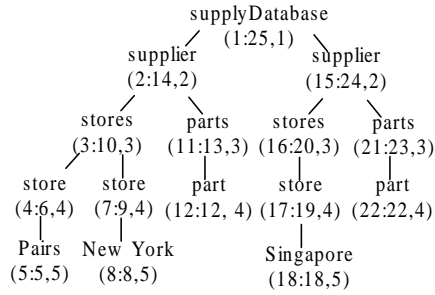


Fig. 2. XML data tree with region encoding

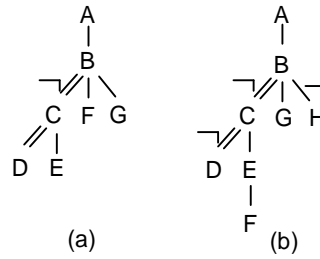


Fig. 3. Two NOT-twigs

Formally, element u is an ancestor of element v iff $startPos(u) < startPos(v)$ and $endPos(u) > endPos(v)$. Similarly, element u is the parent of element v iff $startPos(u) < startPos(v)$, $endPos(u) > endPos(v)$, and $levelNum(u) + 1 = levelNum(v)$.

3.2 NOT-Predicate and Node Operations

Each NOT-twig query has a corresponding tree representation, which contains all the nodes in the query, $\{n_1, n_2, \dots, n_m\}$. Each node n_i and its ancestor (or parent respectively) n_j , are connected by an edge, denoted by $edge(n_i, n_j)$. The tree edges can be classified into one of the following four types: (1) positive ancestor/descendant edge, represented as “//”; (2) positive parent/child edge, represented “|”; (3) negative ancestor/descendant edge, represented as “// -”; (4) negative parent/child edge, represented as “| -”.

A negative edge corresponds to an $edge(n_i, n_j)$ with a not-predicate in XQuery expression. It includes negative parent/child edge and negative ancestor/descendant edge. In this case, node n_j is called a *negative child* of node n_i . Similarly, a positive edge corresponds to an $edge(n_i, n_j)$ without not-predicates in XQuery expression. Node n_j is considered to be a *positive child* of node n_i .

As an example, consider the NOT-twig in Fig. 3(b), $edge(B, C)$ and $edge(C, D)$ are negative ancestor/descendant edge, $edge(B, H)$ is negative parent/child edge. Node B has three children, in which node G is a positive child, node C node H are negative children.

Given a query tree Q , a not-predicate is a subtree in Q such that the edge between the root of the subtree and its ancestor (or parent respectively) is a negative edge. We call the root of the subtree a negative child of its ancestor (or parent respectively).

As an example, the NOT-twig in Fig. 3(a) has one not-predicates (rooted at node C), and the NOT-twig in Fig. 3(b) has three not-predicates (rooted at node C , D and H). The not-predicates are nested, therefore, we can see that not-predicate C contains not-predicate D .

In the following, we define some operations on query tree nodes. $isRoot(n)$, $isLeaf(n)$, and $isOutputNode(n)$ respectively checks if a query node n is a root, a leaf node, or an output node. $is_Neg_Child(n)$ checks if the edge between node n and its ancestor (or parent respectively) has a not-predicate.

$neg_Children(n)$ and $pos_Children(n)$ respectively returns all the nodes that are the negative and positive children of n . $parent(n)$ returns the parent node of n , and the function $children(n)$ gets all child nodes of n . Therefore, we have $neg_Children(n) \cup pos_Children(n) = children(n)$. Function $AD_Neg_Children(n)$ and $PC_Neg_Children(n)$ returns all the negative AD child or negative PC child nodes of n . Similarly, function $AD_Pos_Children(n)$ and $PC_Pos_Children(n)$ returns all the positive AD child or PC child nodes of n .

3.3 Sub-query Matching and the Output Elements

In this paper, a *node* refers to a query node in twig query pattern and *element* refers to a data node in XML data tree. A NOT-twig matching problem can be decomposed into recursive sub-query matching problems.

Given a NOT-twig query Q , a query node n and a XML data tree D , we say that an element e_n (with the tag n) in the XML data tree D **satisfies** the sub-query rooted at n of Q iff:

- (1) n is a leaf node of NOT-query Q ; OR
- (2) For each child node n_i of n in Q :
 - (case i) If n_i is a positive PC child node of n , there is an element e_{n_i} in D such that e_{n_i} is a child element of e_n and satisfies the sub-query rooted at n_i in D .
 - (case ii) If n_i is a positive AD child node of n , there is an element e_{n_i} in D such that e_{n_i} is a descendant element of e_n and satisfies the sub-query rooted at n_i in D .
 - (case iii) If n_i is a negative PC child node of n , there does not exist any element e_{n_i} in D such that e_{n_i} is a child element of e_n and satisfies the sub-query rooted at n_i in D .
 - (case iv) If n_i is a negative AD child node of n , there does not exist any element e_{n_i} in D such that e_{n_i} is a descendant element of e_n and satisfies the sub-query rooted at n_i in D .

We classify the nodes in a NOT-twig query into the following categories:

Definition 1 (output node, non-output node, output leaf node, leaf node). A node n_i in a NOT-twig query is classified as an output node if n_i does not appear below any negative edge; otherwise, it is a non-output node. An output node with no positive children is called a output leaf node. A query node without any children is called a leaf node.

For the NOT-twig in Fig. 3(b), $\{A, B, G\}$ and $\{C, D, E, F, H\}$ are the sets of output nodes and non-output nodes. Note that G is the output leaf node and $\{D, F, G, H\}$ are the leaf nodes.

The output elements for a NOT-twig query is defined in the following:

Definition 2 (Output elements for a NOT-twig Query). *Given an XML document D and a twig query with K output nodes, $\{n_1, n_2 \dots n_k\}$. A tuple $\{e_1, \dots, e_k\}$ is defined to be a **matching answer** for the query iff (1) e_i has the same type (tag name) as n_i ; (2) for each pair of elements e_i and e_j in the tuple, e_j is a descendant (or child respectively) element of e_i in D if $\text{edge}(n_i, n_j)$ is a AD (or PC respectively) edge; and (3) Any output node e_k (with tag K) satisfies sub-query rooted at node k .*

For example, consider the document in Fig. 4(a), we want to match the NOT-twig in Fig. 4(b). $\{< A_1, B_1\}$ is not a matching answer since A_1 doesn't satisfy the sub-query rooted at A since it has a child C_1 that satisfy the sub-query rooted at C . However, $\{A_2, B_2\}$ is a matching answer.

For the NOT-twig in Fig. 4(c), both $\{A_1, B_1\}$, and $\{A_2, B_2\}$ are matching answers, because A_1 is an ancestor of B_1 , A_2 is an ancestor of B_2 , and all of A_1, B_1, A_2, B_2 satisfies the sub-query matching.

4 Negation Twig Join Algorithm

In this section, we present *TwigStackList \neg* , an algorithm for finding all the matching answers of a NOT-twig query against an XML document. We should know that although *TwigStackList \neg* shares similarity with the *TwigStackList* algorithm in the previous work [7], it makes an important extension to handle the NOT-twigs.

4.1 Notation and Data Structures

For each node n in the query twig, a data stream T_n is associated with it. *Stream* is a posting list (or inverted list) accessed by a simple iterator. An XML document is partitioned into streams and an additional region coding label is assigned to each element in the streams. All elements in a stream are of the same tag and ordered by their *startPos*. We can only read the elements in a stream once from head to tail. Cursor c_n to access to the current element in T_n .

Our algorithm uses of two types of data structure: list and stack. For each output node, we associate a list L_n and a stack S_n with it. For each non-output node, only a list L_n is associated with it. The algorithm look-ahead read some elements in input data streams and cache limited number of them to lists L_n in the main memory. At every point during computation: the nodes in stack S_n are guaranteed to lie on a root-leaf path in the database, which means each element is an ancestor or parent of that following it. For each list L_n , a cursor p_n is used to access the element. Similar to stack, elements in each list L_n are also strictly nested from the first to the end.

Based on the data structure definition, we can now define the concept of *head element*:

Definition 3 (head element). *In *TwigStackList \neg* , for each node in the query, if list L_n is not empty, we say that the element indicated by the cursor p_n of L_n is the head element of n . Otherwise, we say that element pointed by c_n in the stream T_n is the head element of n .*

In our algorithm, we use the function $getElement(n)$ to get the *head element* of a query node n .

4.2 Negation Children Extension

We introduce a new concept: *Negation Children Extension* (for short *NCE*), which is important to determine whether an element likely be involves in the results of a NOT-twig query.

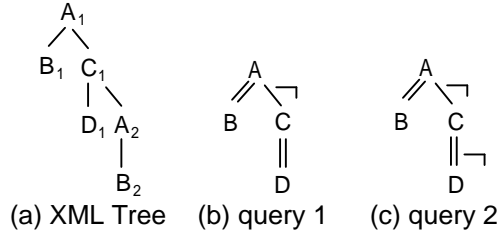


Fig. 4. Illustration of Sub-query Matching

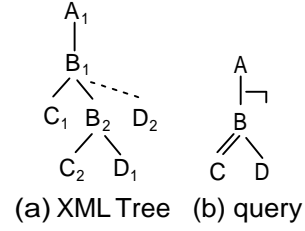


Fig. 5. Sub-optimality example

Given an NOT-twig query Q and a dataset D , we say that an element e_n (tag n) in XML database D has a **Negation Children Extension** (for short *NCE*) based on the following conditions:

1. If in Q , query node n has no positive PC child or it is not an output node, the element e_n has a NCE iff it satisfies the sub-query rooted at n ; OR
2. If in Q , query node n has positive PC child n_i and n is an output node, there is an element e'_n (with tag n) in the path e_n to e_{n_i} such that e'_n is the parent of e_{n_i} and e_{n_i} also has *NCE*. The checking for the positive AD child, negative PC child and AD child nodes remains the same as sub-query matching.

The concept is different from sub-query matching since holistic algorithm cannot guarantee optimality when processing positive PC relationships. When the nodes are output nodes, we can eliminate the useless intermediate results using join operation, similar to the method used in *TwigStackList*. Condition (2) of *NCE* is based on this property. However, if the node is non-output nodes, the positive PC relationship has to be checked before the intermediate results are generated.

For example, consider the XML document and the two queries in Fig. 4(a), (b) and (c). Observe that (1) For both query 1 and query 2, B_1 , B_2 and D_1 has *NCE* since they are leaf nodes. (2) For query 1, C_1 has *NCE*, since D_1 is a descendant of C_1 . However, for query 2, C_1 doesn't have *NCE*. It is because since D_1 is a descendant of C_1 in the XML document, C_1 doesn't satisfy the sub-query rooted at C . (3) For query 1, since C_1 has *NCE*, we can safely say A_1 has no *NCE*. It is because C_1 is a child of A_1 and in the NOT-twig, node C is a negative PC child of node A . A_1 doesn't satisfy the sub-query rooted at A . (4) For query 2, in the XML document, there are no element with tag C that has *NCE*. Also, A_1 has a descendent B_1 with *NCE*. Therefore, A_1 has *NCE*. (5) For both query 1 and query 2, A_2 has *NCE* since A_2 has a descendent B_2 with *NCE*, and does not have any child with the tag C .

In the previous algorithm, both *TwigStack* and *TwigStackList* might output useless intermediate results when a branching node has at least one PC children. Now, we discuss the effect of this sub-optimality problem based on the concept PC-Branching node:

Definition 4 (PC-Branching node). *In a NOT-twig, a node n is called PC-Branching node if n has more than one positive children, among which at least one is a PC child.*

If a PC-Branching node is also an output node, we call it output PC-Branching node. Otherwise, it is called non-output PC-Branching node.

When we match a NOT-twig, the sub-optimality is caused by *PC-Branching* nodes in the query. If the PC-Branching node is an output node, the algorithm can use the method in *TwigStackList* and eliminate the useless intermediate results using join operation.

However, when the PC-Branching node is a non-output node, the useless intermediate result may result in false output. For example, the XML document and query are shown in Fig. 5(a) and (b), the query node B is a non-output PC-Branching node. Initially, the algorithm scans A_1, B_1, C_1 and D_1 . Since only the first element of a stream is read, there is no way for the algorithm to decide if B_1 has a child element D_2 . If the XML dataset does not have element D_2 , the previous methods (*TwigStack* [2] or *TwigStackList* [7]) will still assume that B_1 has a child with tag D . In this case, the element A_1 cannot contribute to the final results and is deleted from the potential solution list. This operation is wrong since we will lose a matching answer A_1 .

Our method to solve the problem of non-output PC-Branching node (of the type n) is to use join operation to find the elements e_n with *NCE* before generating intermediate results. For example, to match the NOT-twig in 5(b), we search for elements of type B that has a descendent element C_i and a child element D_i . Since there is no matching element, we then conclude A_1 is in the matching answers.

In our algorithm, we use the function *isNonBranching(n)* to test if a node n is a non-output PC-Branching node.

4.3 Algorithm: *TwigStackList* \neg

The main algorithm of *TwigStackList* \neg (represented in algorithm 2), which computes answers to a NOT-twig, operates in two phases. In the first phase (line 1-12), the individual query root-leaf paths are output. In the second phase (line 13), these solutions are merged-joined to compute the matching answers to the whole query.

getNext(n) is an important procedure call in the main algorithm of *TwigStackList* \neg . It returns a node n' (possibly $n' = n$). Assume that element $e_{n'}$ is the *head element* of node n' . In our algorithm, the element $e_{n'}$ has *NCE*.

In line 2, if the node is a non-output PC-Branching node, we call *TwigStackListCopy* \neg . This function is a copy of *TwigStackList* \neg but matches the sub-query Q_n (rooted at node n). It uses the same data streams C as the calling function, but terminates when the *getStart(n) > getEnd(Parent(n))*. Therefore, we only want to find the elements that are potential descendants of the element *getElement(Parent(n))*. The final results are inserted to the list of n , L_n .

Line 3-8 check for positive PC child nodes for output nodes (details discussed in *TwigStackList*). Line 9-15 check for negative child nodes of n . We recursively call for every $n_i \in \text{neg}_C\text{children}(n)$. If any returned node g_i is not n_i , we return g_i , if $\text{getStart}(g_i) < \text{getEnd}(n_i)$ (line 9). Otherwise, if the return value is n_i , we *proceed*(n_i) in the main algorithm and call *getNext*(n) again. If $g_i = n_i$ and the *head element* of node n is the ancestor of the *head element* of node n_i (n_i is a negative AD child of n in the NOT-twig), the algorithm concludes the *head element* of n doesn't appear in the final answers (line 16).

Algorithm 1 *getNext*(n)

```

1: if(isLeaf( $n$ )) return  $n$ 
2: if(isNonBranching( $n$ )) TwigStackListCopy-( $Q_n$ .getEnd(Parent( $n$ )))
3: for(each node  $n_i$  in pos_Children( $n$ )) do
4:    $g_i = \text{getNext}(n_i)$ 
5:   if( $g_i \neq n_i$ ) return  $g_i$ 
6:    $\text{Pos}_{n_{max}} = \max_{n_i \in \text{children}(n)} \text{getStartPos}_{n_{max}}$ 
7:    $\text{Pos}_{n_{min}} = \max_{n_i \in \text{children}(n)} \text{getStartPos}_{n_{min}}$ 
8:   while ( $\text{getEnd}(n) < \text{getStart}(n_{max})$ ) proceed( $n$ )
9:   for(each node  $n_i$  in neg_Children( $n$ )) do
10:    while( $\text{getStart}(n_i) < \text{getStart}(n)$ ) proceed  $n_i$ 
11:     $g_i = \text{getNext}(n_i)$ 
12:    if( $g_i \neq n_i$ )
13:      if( $\text{getEnd}(n_i) < \text{getStart}(g_i)$ ) return  $n_i$ 
14:      else return  $g_i$ 
15:    if(is_Neg_AD_Child( $n_i$ ))
16:      while( $\text{getElement}(n)$  is the ancestor of  $\text{getElement}(n_i)$ ) proceed( $n$ )
17:    if ( $\text{getStart}(n) > \text{getStart}(n_{max})$ ) return  $n_{min}$ 
18:    if( $n_{max} \neq \text{null}$ ) MoveStreamToList( $n, n_{max}$ )
19:    for(each node  $n_i$  in  $PC\_Neg\_Children(n)$ ) do
20:      while( $c_n.start < \text{getStart}(n_i)$ ) do
21:        if( $c_n.end > \text{getEnd}(g)$ )  $L_n.append(c_n)$ 
22:        advance( $T_n$ )
23:      if(there is an element  $e_i$  in  $L_n$  such that  $e_i$  is the parent of  $\text{getElement}(n_i)$ )
24:        delete elements  $e_i$  and  $\text{getElement}(n_i)$ 
25:      for(all node  $n_i$  in  $PC\_Pos\_Children(n)$ ) do
26:        if(there is an element  $e_i$  in list  $L_n$  such that  $e_i$  is the parent of  $\text{getElement}(n_i)$ )
27:          if( $n_i$  is the only child of  $n$ ) move the cursor  $p_n$  of list  $L_n$  to point to  $e_i$ 
28:          else return  $n_i$ 
29: return  $n$ 

```

If node n has at least one positive child, line 18 calls *MoveStreamToList* to push elements e_n into the list of node n . Then, line 19-24 check the negative PC relationship in condition (iv) of *NCE*. We push the potential parent of element e_{n_i} into the list of n (line 21). In line 20-21, we make sure the elements are nested from top to end in the list. If we can find an element that is a negative child of an element in the list L_n , the parent element is deleted from the list (line 24). Finally, line 25-28 check the condition (ii) of *NCE*.

Now we discuss the main algorithm of *TwigStackList* \neg (in Algorithm 2). First of all, line 2 calls *getNext* to identify the next element to be processed. If returned node is non-output nodes, the algorithm just proceed the node in line 12. Otherwise, the algorithm check the output nodes the same way as *TwigStackList*.

Example 1. Consider the XML data and query shown in Fig. 4(a) and (b) again. Initially, the stream cursors are pointed to A_1, B_1, C_1 and D_1 . In the first call of *getNext(A)*, the element A_1 is first pushed into the list L_A (in line 18 of *getNext*), then deleted from the list (in line 24 of *getNext*), since C_1 has *NCE* and is a child of A_1 . After the second call of *getNext*, the cursors of A and B are forwarded to A_2, B_2 , and the cursors of C and D are pointing to the end of the stream. The following steps push A_2 to stack and output the intermediate result $\{A_2, B_2\}$. For this query, no merging operation is needed.

Example 2. We now consider the same XML data, but change the query to Fig. 4(c). Initially, the stream cursors points to A_1, B_1, C_1 and D_1 . In the first call of *getNext(A)*, the node C is returned. It is because C_1 is first pushed into the list (in line 18 of *getNext*), then deleted from the list (in line 24 of *getNext*) since leaf node D_1 is a child of C_1 in the XML data. We advance the stream of C and reach the end of the stream. Therefore, in the next call of *getNext*, the element A_1 is pushed to stack and output the path solution $\{A_1, B_1\}$. We call *getNext* again to advance stream A , and push element A_2 into stack S_A . The path $\{A_2, B_2\}$ is then output by the algorithm.

Algorithm 2 *TwigStackList* \neg

```

1: while( $\neg$  end( )) do
2:    $n_{act}$  = getNext(root)
3:   if(! $n_{act}$ .isNonOutputNode( ))
4:     if( $\neg$  isRoot( $n_{act}$ )) cleanParentStack(nact, getStart( $n_{act}$ ))
5:     if(isRoot( $n_{act}$ )  $\vee$   $\neg$  empty(Sparent( $n_{act}$ )))
6:       clearSelfStack( $n_{act}$ , getEnd( $n_{act}$ ))
7:       moveToStack( $n_{act}, S_{n_{act}}, \text{pointertotop}(S_{\text{parent}(n_{act})})$ )
8:       if(isOutputleaf( $n_{act}$ ))
9:         showSolutionsWithBlocking( $S_{n_{act}}, 1$ )
10:      pop( $S_{n_{act}}$ )
11:     else proceed( $n_{act}$ )
12:   else proceed( $n_{act}$ )
13: mergeAllPathSolutions( )

```

4.4 Analysis of *TwigStackList* \neg

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

Lemma 1. *For an arbitrary node n in the NOT-twig query we have $\text{getNext}(n) = n'$. Then the following properties hold:*

1. n' has the NCE

2. Either (a) $n=n'$ or (b) $\text{parent}(n)$ does not have NCE due to the node n'

Lemma 2. Suppose $\text{getNext}(n)=n'$ returns a non-output query node. The head element does not contribute to any matching solutions, since it is not an output node. The Algorithm just proceed the node.

Lemma 3. Any element e that is inserted to stack S_n satisfy the not-predicates of the query. That is, if n has a negative descendant n' in query, then there is no element $e_{n'}$ in stream $T_{n'}$ such that $e_{n'}$ is a descendant of e_n . If element e has a negative PC child e_m (node type m . m is the negative PC child of node n in the query), e is deleted in line 24 of algorithm 1.

Lemma 4. In $\text{TwigStackList}\neg$, when any element e is popped from stack, e is guaranteed not to participate a new solution any longer.

Theorem 1. Given a NOT-twig Q and an XML database D . Algorithm $\text{TwigStackList}\neg$ correctly returns all matching answers for Q on D .

Proof. Using Lemma 2, we know that when getNext returns a query node n in getNext , if the stack is empty, the head element e_n does not contribute to any matching solutions. Thus, any element in the ancestors of n that has positive child e_n with NCE is returned by the getNext before e_n . If n is negative child, we guarantee that each element e_n with NCE in the list L_n is checked to remove their corresponding parent elements by using lemma 3. Furthermore, with lemma 4, we can maintain that, for each node n in the query, the elements that involve in the root-leaf path solution are all in the stack S_n . Finally, each time that $n = \text{getNext}(\text{root})$ is an output leaf node, we output all solution for e_n (line 9 of $\text{TwigStackList}\neg$).

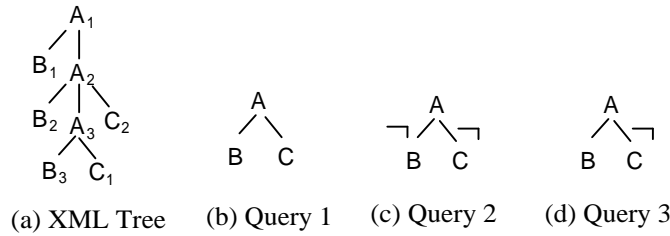


Fig. 6. Optimality study

We now analyze the optimality of $\text{TwigStackList}\neg$. For the Normal-twig join algorithms, TwigStack [2] is optimal for AD only twig patterns; TwigStackList [7] although identifies a larger optimal class than TwigStack , can not guarantee optimality for PC edges in non-branching node. In [3], the author proved that it is difficult to find an optimal Normal-twig pattern matching method, since we cannot determine only from the first elements of various streams if any first element is in the match to a given twig pattern.

However, our algorithm can identify a larger optimal class than *TwigStackList* for NOT-twigs. In particular, the optimality of *TwigStackList*[¬] allows the existence of parent-child relationship in more than one negative branching edges, as illustrated below.

For example, we want to match the NOT-twig in Fig. 6(c) to the dataset in Fig. 6(a). If the naïve method uses *TwigStackList* to solve the problem, we removes the not-predicates and change the query to Fig. 6(b). In order to solve it, *TwigStackList* first scans A_1 , C_1 and B_1 , and pushes element A_1 , A_2 and A_3 into the list L_A . However, since we can only read the head of a stream at a time, when we advance B , we could not decide whether A_1 has a child tagged with C . Therefore, this algorithm will output one useless solution $\{A_1, B_1\}$.

If we use *TwigStackList*[¬] to directly match query 2 (in Fig. 6(c)). We first push element A_1 , A_2 and A_3 into the list L_A . Then, we can immediately identify that A_3 has a child C_1 . Since there is an not-predicate on $edge(A, C)$, A_3 is removed from the list. We advance C and since C_2 is a child of A_2 , A_2 is deleted from L_A . We advance C again, since A_1 doesn't have any child element with the tag name C . We output the path $\{A_1, B_1\}$ as output.

We use the similar method to match the NOT-twig in Fig. 6(d). After we push A_1 , A_2 and A_3 into the stack L_A , we can identify that A_1 has a child B_1 . Since there is an not-predicate on $edge(A, C)$, A_1 is removed from the list. B is advanced to B_2 . Since it is a child of A_2 , A_2 is deleted from the list. B is advanced to B_3 , which is a child of A_3 , A_3 is also deleted from the list. Therefore, there is no matching answers to query 3.

Thus, this example shows that our algorithm may guarantee the optimality for queries with parent-child relationship negative branching edge.

Theorem 2. Consider an XML database D and an NOT-twig query Q without non-output PC-Branching nodes. The worst case I/O complexity of *TwigStackList*[¬] 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 .

5 Experimental Evaluation

We implements two naive twig join algorithms, *naïve-TwigStack* (for short *NTS*) and *naïve-TwigStackList* (for short *NTSL*), to be compared with our algorithm, *twigstacklist*[¬]. The naive methods use the straightforward query decomposition approach. This approach first decomposes the NOT-twig into queries without not-predicates. The decomposed queries are then matched individually (using *TwigStack* [2] or *TwigStackList* [7], respectively) and the NOT-twig solution is calculated by the set-difference of the decomposed query results.

In our experiment, we use the following two metrics to compare the performance of the three algorithms.

- **Intermediate path solutions:** This metric measures the total number of intermediate path solutions. For the naïve methods, the total number is the sum of the intermediate results of all the decomposed queries.
- **Execution time:** We calculate this metric using the average time elapsed to answer a query with ten individual runs.

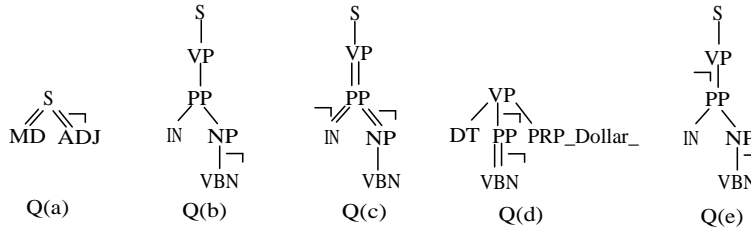


Fig. 7. Five NOT-twigs tested in TreeBank

5.1 Experimental Setup

We use JDK 1.4 with the file system as a simple storage engine. All experiments 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 three real-world and synthetic data sets for our experiments. The first one is a real dataset: TreeBank [11]. The file size is 82M bytes with 2.4 million nodes. The second one is the well-known benchmark data: XMark [14]. The size of file is 115M bytes with factor 1.0. The third one is a Random data set. We generated random uniformly distributed data trees using two parameters: fan-out, depth. We use seven different labels (tag: *a, b, c, d, e, f* and *g*) to generate the data sets.

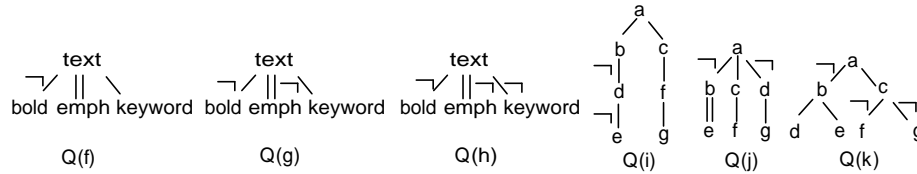


Fig. 8. Six NOT-twigs: Test Q(f), Q(g), Q(h) in XMark; Q(i), Q(j), Q(k) in Random data set

We tested five twig queries (in Fig. 7) in TreeBank, and three twig queries (in Fig. 8) in XMark and Random data set separately. The queries give a comprehensive comparison of the three algorithms, since the queries have different structures and combinations of positive and negative edges.

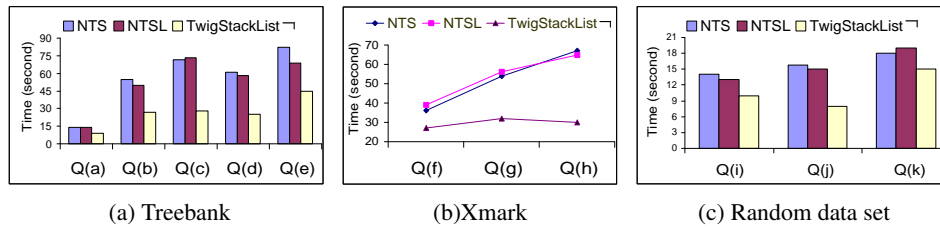


Fig. 9. Execution time of NOT-twig on three datasets

5.2 Performance Study

Fig. 9 shows the results on execution time in the three datasets. We can observe from the three figures that *TwigStackList \neg* is more efficient than the two naïve methods for all the queries. It is because the naïve methods have to match more than one decomposed queries and generate more intermediate results.

An interesting observation is made when we test the queries in XMark database. Query Q(f), Q(g), Q(h) have the same query nodes and structure, but the number of not-predicates is different. We can see from Fig. 9(b) that for *TwigStackList \neg* , the time to match the three queries is almost constant. However, the results for *NTS* and *NTSL* show that when we increase the number of not-predicate, the total execution time increases linearly. It is because as we increase the number of not-predicates, we are increasing the number of decomposed queries that the naïve method need to match.

For each NOT-twig, the number of decomposed queries and the the intermediate results are listed in Table 1. The last column shows the number of path solutions in the matching answers. The results show that *TwigStackList \neg* is sub-optimal if there are PC-Branching nodes in the query, e.g *PP* in Q(b), *VP* in Q(d), *a* in Q(i).

In Table 1, we can see that the number of intermediate results output by our *TwigStackList \neg* is always less than the results output by *NTS* and *NTSL*. It is because when we output the intermediate results in *TwigStackList \neg* , we already considered the not-predicates. Thus, the number of useless intermediate paths is largely reduced.

Query	Dataset	Decomposed Queries	NTS	NTSL	TwigStackList \neg	Useful Solutions
Q(a)	Treebank	2	31197	31197	31143	31143
Q(b)	Treebank	2	64053	61646	60356	58405
Q(c)	Treebank	3	355981	355981	14484	14484
Q(d)	Treebank	3	78857	78675	1789	1508
Q(e)	Treebank	3	215595	209652	78326	67312
Q(f)	XMark	2	181066	171392	22870	21050
Q(g)	XMark	3	228009	224027	12057	12057
Q(h)	XMark	4	308708	306602	7476	7476
Q(i)	Random	3	152	114	58	36
Q(j)	Random	3	1701	1461	138	138
Q(k)	Random	4	1731	1120	837	436

Table 1. The number of intermediate path solutions

Therefore, according to the experimental results, we can conclude that our new algorithm *TwigStackList \neg* could be used to evaluate twig pattern with not-predicates because it has obvious performance advantage over the straightforward approaches: *NTS* and *NTSL*. *TwigStackList \neg* guarantees the I/O optimality for a large query class.

6 Conclusion and Future Work

In this paper, we proposed a new holistic twig join algorithm, called *TwigStackList \neg* , to process NOT-twig query. Although holistic twig join has been proposed to solve

Normal-twig patterns, applying it to NOT-twig matching is nontrivial. We developed a new concept *Negation Children Extension* to determine whether an element is in the results of a NOT-twig query. We also make the contribution by identifying a large query class to guarantee I/O optimality for *TwigStackList* \rightarrow . The experimental results show that our algorithm is more effective and efficient than the naïve method.

In the future, we will improve the algorithm based on the following two issues: one is to design an efficient index scheme that might change the format of the input data. Another possible issue to improve our algorithm is to identify a larger optimal query class for NOT-twig matching.

References

1. Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, pages 141–152, 2002.
2. Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of SIGMOD*, pages 310–321, 2002.
3. Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proceedings of SIGMOD*, 2005.
4. Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of twig queries with or-predicates. In *In Proceeding of SIGMOD*, pages 59–70, 2004.
5. Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *In Proceeding of VLDB*, pages 273–284, 2003.
6. Enhua Jiao, Tok Wang Ling, Chee Yong Chan, and Philip S. Yu. Pathstack \rightarrow : A holistic path join algorithm for path query with not-predicates on XML data. In *Proceedings of DASFAA*, 2005.
7. Jiaheng Lu, Ting Chen, and Tok Wang Ling. Efficient processing of XML twig patterns with parent child edges: A look-ahead approach. In *Proceedings of CIKM*, pages 533–542, 2004.
8. Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In *In Proceeding of VLDB*, 2005.
9. Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, and Wei Ni. Efficient processing of ordered XML twig pattern. In *In Proceeding of DEXA*, 2005.
10. Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using präfer sequences. In *Proceedings of ICDE*, 2004.
11. Treebank. <http://www.cs.washington.edu/research/xmldatasets/www/repository.html>.
12. Haixun Wang and Xiaofeng Meng. On the sequencing of tree structures for XML indexing. In *In Proceeding of ICDE*, pages 372–383, 2005.
13. Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A dynamic index method for querying XML data by tree structure. In *Proceedings of SIGMOD*, 2003.
14. XMARK. <http://monetdb.cwi.nl/xml>.
15. XPath. <http://www.w3.org/TR/xpath>.
16. Chun Zhang, Jeffery Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, pages 425–436, 2001.