Efficient Processing of Complex Twig Pattern Matching

Wei Wang, Jinqing Zhu, Xiaofeng Meng

Renmin University of China, Beijing, China
{dawei,zhujinqing,xfmeng}@ruc.edu.cn

***Abstract*—As a de facto standard for information representation and exchange over the internet, XML has been used extensively in many applications. And XML query technology has attracted more and more attention in data management research community. Standard XML query languages, e.g. XPath and XQuery, use twig pattern as a basic unit to match relevant fragments from a given XML document. However, in most existing work, only simple containment relationships are involved in the twig pattern, which makes it infeasible in many cases. In this paper, we extend the original twig pattern to Complex Twig Pattern (CTP), which may contain ordered relationship between query nodes. We give a detailed analysis of the hard nuts that prevent us from finding an efficient solution for CTP matching, and then propose a novel holistic join algorithm, LBHJ, to handle the CTP efficiently and effectively. We show in experimental results that LBHJ can largely reduce the size of intermediate results and thus improve the query performance significantly according to various metrics when processing CTP with ordered axes.**

I. INTRODUCTION

As a de facto standard for information representation and exchange over the internet, XML has been used extensively in many applications. Query capabilities are provided through twig pattern queries, which are the core components for standard XML query languages, e.g. XPath [2] and XQuery [3]. A twig pattern query can be naturally represented as a node-labeled tree, in which each edge represents either *Parent-Child* (***P-C***) or *Ancestor-Descendant* (***A-D***) relationship. For example, the twig pattern query written in XPath [2] format, Q: A[B]//D, selects elements D, which is a descendant of A whose child elements include B.

Besides the *A-D* and *P-C* relationship, XPath also supports four ***ordered axes***: following-sibling, preceding-sibling, following and preceding. While researchers have proposed many holistic twig join algorithms [4,7,9,5,10] to efficiently find all the occurrences of a twig pattern from an XML database, a key problem of these existing works that has been largely ignored is that they can not handle *ordered* XML twig query efficiently which contains ordered relationship between query nodes. We call such query pattern containing ordered axes ***Complex Twig Pattern*** (***CTP***).

Naive method of CTP processing consists of three steps, (1) split a CTP into several simple twig queries which do not contain ordered relationship, (2) evaluate each twig query separately using existing state-of-the-art twig join algorithms, (3) use an ordered structural join method to merge the partial results together to get the final results. We call this method decomposition based approach. Although existing twig join algorithms can efficiently process a twig pattern query without ordered axis, when they are used to process a CTP query, large amount of useless intermediate path solutions may be produced because the decomposed simple twig queries are unaware of the ordered structural constraint that existed between them. And performance of the decomposition based approach is largely determined by the size of intermediate path solutions. Under extreme circumstance, the performance will degrade significantly, which makes the decomposition based approach infeasible for CTP query processing.

Consider CTP query Q1 in Figure 2 (a) and the XML document in Figure 1 as an example, the naive method will split Q1 into two twig queries, //A//C and //A//D, which will return 11 intermediate path solutions including (a_1, c_1), (a_1, c_2), …, (a_1, c_8), (a_1, d_1), (a_1, d_2), (a_1, d_3), and we can easily figure out that most of them are useless. OrderTJ [11] can not handle this kind of *Ordered* query efficiently because it needs to translate Q1 to //A//*[C]/D, where * denotes any tag, and the children of *, C and D, should keep the right documental order, thus it needs to scan all element streams, which is time-consuming and inefficient. Moreover, it cannot process the following two kinds of queries: (1) query patterns which have not been specified with a root node, e.g. //A/following-sibling::B, and (2) query patterns with mixed order and unordered relationships, e.g. Q2 in Figure 2 (b).

The low efficiency of existing methods lies in the fact that they ignore the correlation between query nodes connected by following-sibling relationship in a CTP, thus lots of useless intermediate path solutions will be produced.

Aware of these pitfalls, we propose a novel query processing method named LBHJ for CTP query processing. Our method processes a CTP in a holistic way without decomposing it into several simple twig patterns and then processing each one separately, thus our method produce much less intermediate path solutions than the naive method. Consider Q1 in Figure 2 (a) and the XML document in Figure 1 again, our method only produces 3 intermediate path solutions, i.e. (a_1, c_1, d_3), (a_1, c_4, d_3), (a_1, c_5, d_2).

Our contributions can be summarized as follows:

- We give a detailed analysis of the following-sibling relationship, and present some conclusions that can be used to avoid the redundant operations when evaluating a CTP.
- We propose a novel data structure, Level Buffer, to cache some temporal results, and then we propose a new holistic join algorithm LBHJ, which can be used to

find final answers of the given CTP in a holistic way using Level Buffer.

- We analyse the performance of the LBHJ algorithms and compare it with the decomposition based approach. The experimental results with datasets of various characteristics demonstrate that our method can work very efficiently in terms of various evaluation metrics.

The rest of this paper proceeds as follows. Section 2 is dedicated to some background knowledge and problem definition. A naive solution for answering CTP query is presented in section 3. In section 4, we analyse the following-sibling relationship and give some guidelines for buffering elements. In section 5, we present our main algorithm, LBHJ, and the detailed analysis of LBHJ. We report our experimental results in Section 6. In Section 7, we discuss some related work. Finally, we conclude our contribution and future work in Section 8.

II. Background

A. Data Model and Labeling scheme

An XML document can be modelled as a rooted, node-labelled tree, where nodes represent elements, attributes and text data while edges represent direct nesting relationships between nodes in the tree. Formally, tree *T* can be represented as a tuple, $T=(V,E,\Sigma,M,r)$, where *V* is the node set, $E \subseteq V \times V$ is the edge set, Σ is an alphabet of labels and text values, $M: V \rightarrow \Sigma$ is a function that maps each node to its label, and $r \in V$ is the root node in *T*. Figure 1 shows the tree representation of a sample XML document.

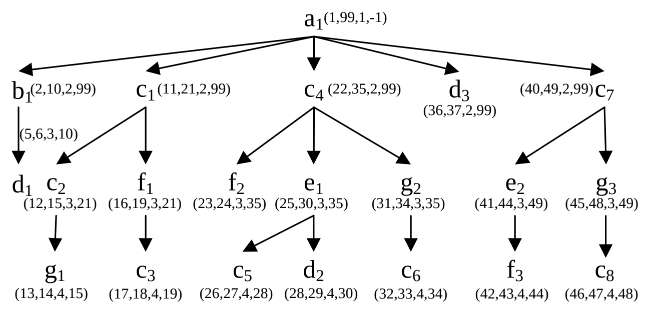


Fig. 1 A sample XML document

Most existing XML query processing algorithms rely on a positional representation of elements, e.g. region encoding scheme [18], thus each node in the given XML document is labelled as a triple (Start, End, Level) based on its left to right deep-first traversing the document. An element *u* is an ancestor of another element *v* iff. *u*.Start < *v*.Start < *u*.End. *u* is the parent of *v* iff. *u*.Start < *v*.Start < *u*.End and *u*.Level = *v*.level-1. Moreover, given two elements *u* and *v*, *u* appears before *v* in document order iff *u*.Start < *v*.Start, which is denoted as *u* << *v*.

Although region encoding can be used to check the containment relationship between two elements in constant time, but when considering ordered relationship, it does not work efficiently anymore. This is because the labels of two elements with the form (Start, End, Level) do not carry enough information for determining sibling relationship. While Dewey ID [14] can be used to judge whether two elements have sibling relationship, this method doesn't work efficiently enough in practice since each Dewey ID is complex compared with region encoding. *In this work, we extend region encoding scheme by adding one additional field, thus each element is labelled using a tuple in the form: (Start, End, Level, ParEnd), where ParEnd denotes the End value of its parent node. Given two elements u and v, if u.ParEnd = v.ParEnd, then they have sibling relationship. Further more, if v.Start > u.Start, then v is a following sibling of u. As shown in Figure 1, each element is labelled using our extended region encoding scheme.*

B. Complex Twig Pattern Matching and Problem Definition

As a CTP can contain all kinds of axes, it is hard to consider all of them simultaneously, Olteanu et al. [12, 13] showed that using special rules, XPath queries with reverse axes can be equivalently rewritten as a set of twig pattern queries without reverse axes, thus the core problem is how to efficiently find the desired results of these transformed queries. *In this paper, we focus on the query evaluation of query pattern with containment relationship and following-sibling relationship*, which can be denoted as $P^{[/,//,\rightarrow]}$, where '/' denotes *P-C* relationship, '//' denotes *A-D* relationship, and '→' denotes *following-sibling*(*F-S*) relationship, respectively. In the following sections, we use CTP to denote query pattern containing only *P-C*, *A-D* or *F-S* edges.

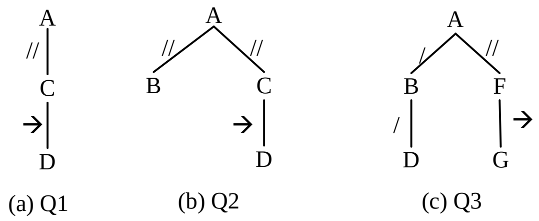


Fig. 2 Three complex twig patterns

Matching a CTP against an XML database is to find all occurrences of the pattern in the database. Formally, given a CTP query *Q* and an XML database *D*, a match of *Q* in *D* is identified by a mapping from nodes in *Q* to element nodes in *D*, where: (i) the query node predicates are satisfied by the corresponding database elements; (ii) the structural relationships (i.e. *P-C*, *A-D* or *F-S*) between query nodes are satisfied by the corresponding database elements. The answer to query *Q* with *n* nodes can be represented as a *n*-array tuple (e_1, e_2, …, e_n) which identifies a distinct match of *Q* in *D*. Consider the CTP query in Figure 2 (b) and the XML document in Figure 1. One match of it is (a_1, b_1, d_1, f_2, g_2).

III. Naive Solution

In this section, we present a naïve solution shown in Algorithm 1 to process a given CTP. This method is a simple extension of the TwigStack algorithm, which considers a CTP as several simple twig pattern queries connected by *F-S* edges. It consists of two steps: (1) process each twig pattern query separately to get the intermediate results (line 1-3), (2) join the intermediate results to get the final answers (line 4).

```
Algorithm 1 TwigStack + Join(Q)
1:  Decompose Q into a set of twig patterns S
2:  For each TwigPattern Q_i in S
3:      Output intermediate results of Q_i using TwigStack(Q_i)
4:  Merge all intermediate results to get final answers
```

EXAMPLE 1: Consider Q3 in Figure 2 (c) and the XML document in Figure 1. Using Algorithm 1, we need to discompose it into two twig patterns, i.e. //A[B/D]//F and //A//G. After processing each twig pattern separately using TwigStack algorithm, two sets of intermediate results will be produced. The first set of intermediate results consists of four path solutions, i.e. (a_1, b_1, d_1), (a_1, f_1), (a_1, f_2), (a_1, f_3). The second set consists of three path solutions, i.e. (a_1, g_1), (a_1, g_2) and (a_1, g_3). In line 4, all these path solutions are merged together to get the final answers. In fact, however, only (a_1, b_1, d_1), (a_1, f_2) and (a_1, g_2) are useful.

The problem of this method is that large amount of useless intermediate results may be produced since it doesn't consider *F-S* relationship contained in the query itself in the first step.

IV. Analysis of Following-Sibling Relationship

A. Why Caching Elements

For the CTP query Q4: //*C*/following-sibling::*D* and the XML document in Figure 1, the structural relationship of elements with tag *C* and *D* is shown in Figure 3, where each line denotes the two nodes that connected by it have *F-S* relationship.

Simple join without caching elements needs to visit some elements several times. For example, in Figure 3, assume cursor C_C points to elements with tag C, C_D points to elements with tag D and C_C moves according to the position of C_D, if C_D points to d2, C_C need to move from c_1 to c_5 to join with d_2. After C_D moves to d3, C_C needs to backtrack from c_5 to c_1 so as to join with d_3, so this method needs to process many elements multiple times. Intuitively, for the *F-S* relationship, a simple merge join solution may have to process large amount of elements more than once. Therefore, if we want to avoid this case, it is inevitable to cache some elements.

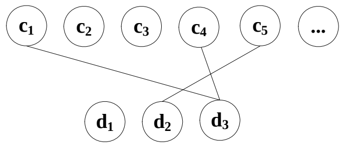


Fig. 3 An example of following-sibling relationship

B. Buffering Guidelines:

As shown in Figure 4 (a), for *u* and its parent node parent(*u*)[1], let D(*u*) denote all u's descendant elements, L(*u*) (R(*u*)) denote the descendant elements of parent(*u*) which appear before (after) *u*, then preceding sibling and following sibling elements can be found in L(*u*) and R(*u*), respectively. For each element *v* in L(*u*), we have parent(*u*).Start < *v*.Start < *u*.Start, similar property can be got for elements in R(u).

Consider CTP query //P/following-sibling::F and the XML document in Figure 4 (b). Assume the current element with tag P is p_1, we use C_F to denote the current processed element with tag F, and further we assume $p_1 << C_F$. Then we have to consider three cases for C_F:

a. C_F equals to f_1 or f_3, which is the following sibling of p_1. i.e. p_1.Start < f_1.Start and p_1.ParEnd = f_1.ParEnd (same condition holds for f_3)).
b. C_F equals to f_2, which is an element in R(p_1), and not a following sibling of p_1. Note that although f_2 is not a following sibling of p_1, maybe there is another element that is a following sibling of p_1, e.g. f_3 in Figure 4 (b), so p_1 should be cached for further processing. In this case p_1.Level < f_2.Level, p_1.Start < f_2.Start < parent(p_1).ParEnd.
c. C_F equals to f_4 and is not in R(p_1), which means that p_1 does not match f_4, so we can safely discard p_1 and process the next element with tag P. In this case p_1.ParEnd < f_4.Start.

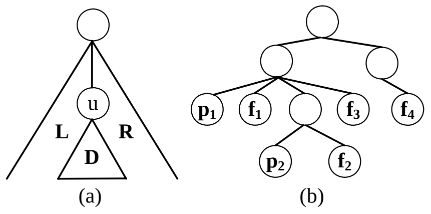


Fig. 4 following-sibling relationship in XML document

LEMMA 1: Given three elements *u*, *v* and *w*, assume that *u* << *v* << *w*. If *u*.ParEnd = *w*.ParEnd, then *u*.level = *w*.Level and *v*.Level ≥ *u*.Level.

Proof: According to the condition *u*.ParEnd = *w*.parEnd, we have *u*.Level = *w*.Level. Assume element *x* is the parent of *u* and *w*, then *v* can appear in four regions, which are shown in Figure 4(a), i.e. D(*x*), A(*x*), L(*x*) and R(*x*), where A(*x*) contains all the ancestor elements of *x*. Because *u* and *w* are children of *x*, so *v* cannot appear in L(*x*) and R(*x*) since they corresponds to *v* << *x* << *u* << *w* and *x* << *u* << *w* << *v*, respectively, which conflicts with the assumption. If *v* appears in A(*x*), we have *v* << *x* << *u* << *w*, also contradict with *u* << *v* << *w*. Then *v* can only appear in D(*x*), where we have *x*.Level + 1 = *u*.Level = *w*.Level ≤ *v*.Level.

EXAMPLE 2: Consider the XML document in Figure 5, f_2 and g_2 have the same parent c_4, and $f_2 << g_2$. Assume a_1.Level = 1. According to the documental order, for any element *x* that satisfies $f_2 << x << g_2$, then $x \in \{e_1, c_5, d_2\}$, so *x*.Level ≤ 3, and f_2.Level = g_2.Level = 3.

THEOREM 1: Consider the CTP query //P/following-sibling::F. Assume elements p_1 and p_2 has tag P, element *f* has tag F. All the three elements satisfy that $p_1 << p_2 << f$, and p_1.Level > p_2.Level, then p_1 cannot be a preceding sibling of *f*, and p_1.ParEnd != *f*.ParEnd.

[1] parent(*u*) is used to get the parent node of *u*.

Proof: Assume that p_1 and f have *F-S* relationship, then $p_1 << f$ and p_1.Level = f.Level, according to Lemma 1, we have p_2.Level ≥ p_1.Level, which contradicts the given condition p_1.Level > p_2.Level.

EXAMPLE 3: Consider the XML document in Figure 5 and the CTP query //C/following-sibling::D, assume the cursor of D points to d_3. Since $c_2 << c_4 << d_3$ and c_2.Level > c_4.Level, according to THEOREM 1, we know that c_2 cannot be a preceding sibling of d_3. Moreover, since $c_3 << c_4 << d_2$ and c_3 is at the same level with d_2, according to THEOREM 1, we can also know that c_3 cannot be a preceding sibling of d_2.

THEOREM 2: Given three elements u, v and w ($u << v << w$), u.Level = v.Level, if u.ParEnd != v.ParEnd, then u cannot be a preceding sibling of w.

The intuition is obvious, if u is a preceding sibling of w, then according Lemma1, v must be a descendant of parent(u), since u.Level = v.Level, then u.ParEnd = v.ParEnd. This conflicts with the condition of u.ParEnd != v.ParEnd, so u cannot be a preceding sibling of w.

EXAMPLE 4: Consider the XML document in Figure 5 again. Since $f_1 << f_2 << g_2$, f_1.Level = f_2.Level and they have different parents, i.e. c_1 and c_4, respectively. According to THEOREM 2, g_2 cannot be a following sibling of f_1.

Based on the above analysis, we introduce a new data structure, Level Buffer (LB), to efficiently process CTP queries. A LB is a chain of linked list, among which each list contains elements that have the same Level value. Moreover, we introduce the following rule to guide the buffering of elements.

Rule 1: Given two elements u, v in the LB such that $u << v$, then they must satisfy at least one of the following two conditions:

a. u.Level = v.Level and u.ParEnd = v.ParEnd
b. u.Level < v.Level

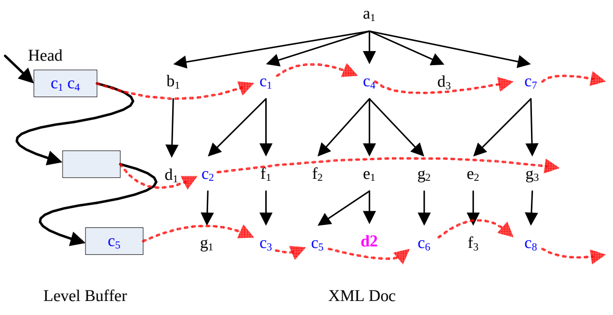


Fig. 5 An example of Level Buffer containing elements with tag C

EXAMPLE 5: Consider the XML document in Figure 5 and the CTP query //C/following-sibling::D. Initially, cursors C_C and C_D point to c_1 and d_1, respectively. Since $d_1 << c_1$, C_D is moved to d_2 and c_1 is pushed into LB, then C_C is moved to c_2, then c_2 and c_3 are pushed into LB since their Level value are larger than that of c_1 and they appear before d_2 in documental order. After that, C_C points to c_4, then we can safely discard c_2 and c_3 since the Level value of c_4 is less than that of c_2 and c_3. After c_4 and c_5 are pushed into LB, the status of LB is shown in Figure 5.

V. Level Buffer based Holistic Join Algorithm

A. *Notation and Data Structure*

In our method, each query node q in a CTP is associated with a LB$_q$, a cursor C_q and a data stream T_q. C_q can point to some elements in T_q , especially, we say C_q is NULL if all elements in T_q are processed, and C_q is also used to denote the element it points to. Before the algorithm is executed, all cursors point to the first elements in each data stream. We can use Advance(C_q) to make C_q point to the next element. The self-explaining functions isRoot(q) and isLeaf(q) are used to determine whether q is a root node or a leaf node. The function children(q) is used to return all the child nodes of q and parent(q) is used to return the parent node of q.

What should be noticed is that each element e_q in LBq has two pointers, one is e_q.pSelfA, which points to the ***nearest ancestor element*** that has the same tag in LB$_q$, the other is e_q.pStrucA, which points to the ***nearest element*** $e_{\text{parent}(q)}$ in LB$_{\text{parent}(q)}$, this element satisfies the structural constraint of parent(q) and q. Obviously, we can use the first pointer to maintain the property of stack. Using the two pointers, we can make full use of the benefits of Level Buffer and stack, thus the *A-D* and *F-S* relationship can be processed elegantly using Level Buffer.

DEFINITION 1 (*Possible Solution Extension* (PSE)): Let Q be a CTP, we say a query node q of Q has a PSE iff q satisfies any one of the following conditions:

1) isLeaf(q) ∧ $C_q \neq$ NULL, or
2) for each $q' \in$ children(q)
   (i) $q//q'$ ∧ hasPSE(q') ∧ $C_q//C_{q'}$[2], or
   (ii) $q \rightarrow q'$ ∧ hasPSE(q') ∧ ($C_q << C_{q'}$)[3]

We use PSE to guide the execution of getNext() in our method. Intuitively, a query node q has a PSE means that all current elements corresponding to nodes that have *A-D* relationship with q satisfy the structural constraints of the sub-tree rooted at q, and all elements corresponding to nodes that have *F-S* relationship with q satisfy that C_q appears before them in documental order.

B. *Algorithm LBHJ*

Algorithm 2, LBHJ, operates in two phases. In the first phase (line 1-11), getNext(Q) is called repeatedly (line 2) to get a query node q with PSE. If q is not the root node, then we need to pop all elements from LB$_{\text{parent}(q)}$ that are useless according to C_q, which is further classified into two cases: (1)

[2] $q// q'$ means q and q' have *A-D* relationship, $C_q//C_{q'}$ means C_q is ancestor of $C_{q'}$. hasPSE(q) checks whether q has a Partial Solution Extension.

[3] $q \rightarrow q'$ means q and q' have *F-S* relationship, $C_q \rightarrow C_{q'}$ means $C_{q'}$ is a following sibling of C_q, i.e. C_q and $C_{q'}$ satisfy the structural constraint of $q \rightarrow q'$.

parent(q)//q, then all elements that are not ancestor of C_p will be popped from LB$_q$. (2) parent(q)→q, then all elements that have larger Level value than C_p will be popped from LB$_q$. In line 5, if q is root node or C_q has matched elements in LB$_{parent(q)}$, then after popping all unmatched elements from LB$_q$ (line 6), C_q will be pushed into LB$_q$ (line 7). If q is a leaf node, the path solutions related with C_q will be produced using the blocking technique proposed in [4] (line 8-9). Then C_q is moved to the next element (line 10). In the second phase, all produced path solutions are merged together to get the final answers (line 12). Note that in Procedure Push(), nearestAnc(LB$_q$, C_q) is used to get the lowest ancestor of C_q from LB$_q$, and nearestEle(LB$_{parent(q)}$, C_q) is used to get the nearest element according to position relationship in documental order that satisfies the structural constraint of parent(q) and q.

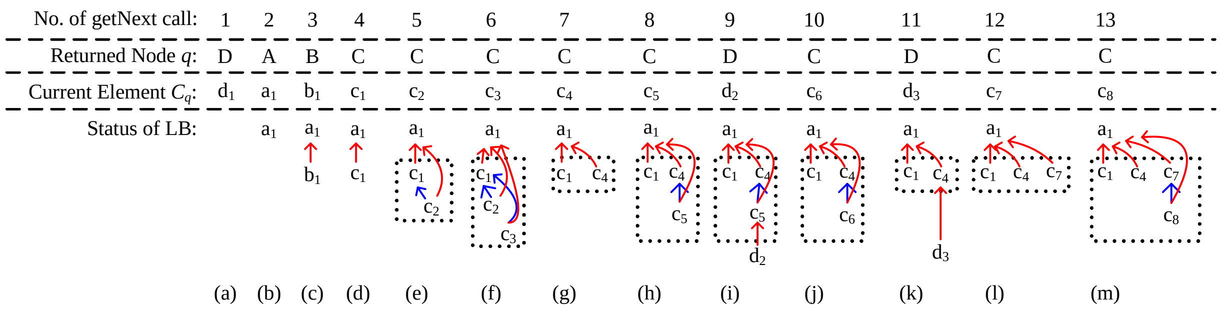
```
Algorithm 2 LBHJ(Q)        // Q is a CTP
 1: while (!end($Q$)) do
 2:    $q$ = getNext($Q$);
 3:    if not isRoot($q$) then
 4:       cleanLB(LB$_{parent(q)}$, $C_q$)
 5:    if isRoot($q$) or hasMatchedEle(LB$_{parent(q)}$, $C_q$) then
 6:       cleanLB(LB$_q$, $C_q$)
 7:       Push(LB$_q$, $C_q$)
 8:       if isLeaf($q$) then
 9:          outputPathSolutionsWithBlocking($C_q$)
10:    Advance($C_q$)
11: end while
12: MergeAllPathSolutions();
Procedure cleanLB(LB$_q$, $C_p$)
 1: if $q//p$ then
 2:    Pop all elements that are not ancestor of $C_p$ from LB$_q$
 3: if $q \rightarrow p$ then
 4:    Pop all elements that have larger Level value than $C_p$ or
       elements that have same Level value but not same parent
       with $C_p$ from LB$_q$
Function hasMatchedEle(LB$_q$, $C_p$)
 1: if $\exists e \in$ LB$_q$($q//p \wedge e//C_p$) then return TRUE
 2: if $\exists e \in$ LB$_q$($q \rightarrow p \wedge e \rightarrow C_p$) then return TRUE
 3: return FLASE
Procedure Push(LB$_q$, $C_q$)
 1: $e$ = nearestAnc(LB$_q$, $C_q$)
 2: if $e$!=NULL then $C_q$.pSelfA = $e$
 3: else $C_q$.pSelfA = NULL
 4: e= nearestEle(LB$_{parent(q)}$, $C_q$)
 5: if isRoot($q$) then $C_q$.pStrucA = NULL
 6: else $C_q$.pStrucA = $e$
```

What should be noticed in LBHJ is that the conclusions we got from Section 4 are applied in cleanLB(), which will largely reduce the number of buffered elements at running time. Moreover, since all elements in LB are organized according to their Level value, nearestAnc(), nearestEle() and hasMatchedEle() can be executed in constant time. So it is easily to understand that our method will achieve similar performance for CTP with only *A-D* edges.

getNext(), as shown in Algorithm 2, is the core function called in LBHJ, in which we need to consider *A-D* and *F-S* relationship simultaneously. getNext() is used here to get a query node with a PSE, from which we can get an element that may participate in final answers. If q is a leaf node, it will be returned directly in line 1. If not, however, in line 2-4, for each child p of q, if p' (returned by getNext(p)) is not equal to p, then p' is returned in line 4. If all children of q have PSEs, then we need to determine whether q has a PSE. In line 5-6, we find n_{min}, and n_{max} which have the minimal and maximal Start value from all children that has *A-D* relationship with q. In line 7, we find r_{min}, which has the minimal Start value from all children that has *F-S* relationship with q. In line 8-9, C_q is forwarded until C_q.End is not less than C_{nmax}.Start. If C_q.Start is larger than C_{nmin}.Start, n_{min} is returned in line 10. In line 11, if C_q.Start is larger than C_{rmin}.Start, n_{min} is returned. At last, if all children of q can satisfy the structural constraints with q, q is returned with a PSE in line 12.

```
Algorithm 3 getNext(q)
 1: if isLeaf($q$)=TRUE then return $q$
 2: for $p \in$ children($q$) do
 3:       $p$' = getNext($p$)
 4:       if $p' \neq p$ then return $p$'
 5: $n_{min}$=minarg$_p${$C_p$.Start | $q//p$}
 6: $n_{max}$=maxarg$_p${$C_p$.Start | $q//p$}
 7: $r_{min}$ = minarg$_{qi}${$C_p$.Start | $q \rightarrow p$ }
 8: while ($C_q$.End < $C_{nmax}$.Start) do
 9:       Advance($C_q$)
10: if $C_q$.Start > $C_{nmin}$.Start then return $n_{min}$
11: if $C_q$.Start > $C_{rmin}$.Start then return $r_{min}$
12: return $q$
```

EXAMPLE 6: Consider Q2 in Figure 2 (b) and the XML document in Figure 1. All returned query nodes, the elements pointed by cursors of these query nodes and the status of LB are presented in Figure 6. Initially, the four cursors C_A, C_B, C_C and C_D point to a_1, b_1, c_1 and d_1, respectively. The first call of getNext(A) returns D with a PSE, since C and D have *F-S* relationship, and d_1 appears before c_1, thus d_1 is discarded directly and then C_D moves to d_2. The second call of getNext(A) returns A with C_A pointing to a_1, since A is root node, a_1 is pushed into LB$_A$, then all elements with tag A are processed and C_A equals to NULL. The third call of getNext(A) returns B with C_B pointing to b_1, since b_1 has a matched element in LB$_A$, i.e. a_1, b_1 is pushed into LB$_B$, the current LB status is shown in Figure 6 (c). The fourth to eighth call of getNext(A) returns C with C_C points to c_1, c_2, c_3, c_4, c_5, respectively. As shown in Figure 6 (d-h), each element in Level Buffer has two pointers, one is pSelfA, shown in Figure 6 as blue arrows, and the other is pStrucA, shown in Figure 6 as red arrows. pSelfA is used to maintain the *A-D* relationship between elements in the same LB with the same tag, and pStrucA is used to maintain the structural relationship between elements with different tag name. The next call of getNext(A) returns D with C_D pointing to d_2, as shown in Figure 6 (i), d_2 will be pushed into LB$_D$, and d_2.pStrucA points to c_5. The tenth call of getNext(A) returns C with C_C pointing to c_6, since the parent of c_5 is not equal to that of c_6, c_5 is popped from LB$_C$, then c_6 is pushed into LB$_C$, the current status is shown in Figure 6 (j). The next call of getNext(A) returns D with C_D pointing to d3, since C and D have *F-S* relationship and the Level value of d_3 is less than that of c_6, c_6



No. of getNext call: 1 2 3 4 5 6 7 8 9 10 11 12 13
Returned Node q: D A B C C C C C D C D C C
Current Element Cq: d1 a1 b1 c1 c2 c3 c4 c5 d2 c6 d3 c7 c8
Status of LB:
(a) (b) (c) (d) (e) (f) (g) (h) (i) (j) (k) (l) (m)

Fig. 6 The demonstration of running example

is popped from LB_C, then d_3 is pushed into LB_D, the current status is shown in Figure 6 (k). The remainder two calls of getNext(A) is similar to the above description and we omit for limited space. Note that the intermediate path solutions are output when an element of leaf node is pushed into a LB, and the output strategy is similar to that of TwigStack [4] using blocking technique. The intermediate path solutions are (a_1, b_1), (a_1, c_1, d_3), (a_1, c_4, d_3) and (a_1, c_5, d_2). In the second phase of LBHJ, all the four path solutions are merged together to get the final answers, they are (a_1, b_1, c_1, d_3), (a_1, b_1, c_4, d_3) and (a_1, b_1, c_5, d_2).

When *P-C* edges appear in the given CTP, we just need to take the level information of each element into account, the detailed description is omitted in Algorithm 2 for simplicity.

C. *Analysis of LBHJ*

We first show the correctness of LBHJ and then analyse the complexity of LBHJ. For simplicity, we say an element is useful if it can participate in at least one final answer.

LEMMA 2: Any useful element C_q will be pushed into LB_q, and if C_q can be pushed into LB_q, it must satisfy the structural constraint with an element in $LB_{parent(q)}$ (except that q is the root node)

Proof: From the discussion about getNext() we know that all elements that are possible useful are returned by getNext(Q), and for each returned element C_q, if there exists an element $e_{parent(q)}$ in $LB_{parent(q)}$ that satisfies the structural relationship of <parent(q), q> with C_q (line 5 in Algorithm 2), then C_q will be pushed into LB_q (line 8 in Algorithm 2).

THEOREM 3: Let Q be a CTP and D be an XML document, Algorithm LBHJ correctly returns all answers for Q on D.

Proof: When an element C_q is pushed into LB_q, we need to modify two pointers, i.e. e_q.pSelfA and e_q.pStrucA, the former points to its ***nearest ancestor element*** in LB_q, it is used to maintain the *A-D* relationship among elements in the same LB, thus we can also use LB as a stack when no *F-S* relationship related with q; the other pointer points to the ***nearest element*** $e_{parent(q)}$ in $LB_{parent(q)}$, which satisfies the structural constraint of parent(q) and q with C_q, the structural constraint may be *A-D* or *F-S* relationship. If q is a leaf node, all path solutions related with C_q is produced in line 10 of Algorithm 2. Although some path solutions may be useless, all useful path solutions are produced through this operation. In the second phase (line 13 in Algorithm 2), all these path solutions are merged to compute the final answers. Thus we know that Algorithm LBHJ correctly returns all answers for Q on D.

THEOREM 4: Let Q be a CTP query and D be an XML document, the worst case space complexity of Algorithm LBHJ is $O(|Q|*H_{doc}*Fanout_{doc})$, where $|Q|$ denotes the size of Q, H_{doc} denotes the maximal height of D and $Fanout_{doc}$ denotes the maximal fan out of the elements in the document, the worst case time complexity of Algorithm LBHJ is $O(Input\_Data\_Size * |Q| + Inter\_Result\_Size + Output\_Result\_Size)$.

The proof is obvious in worst case. Level buffer will cache alll processed elements, whose size equals to $|Q|*H_{doc}*Fanout_{doc}$. Theoretically, the worst case space complexity is large, however, usually in practice, $|Q|$, H_{doc} and $Fanout_{doc}$ are small enough, moreover, we have propose several theorem in Section 4 to reduce the size of buffered elements, such that only a small fraction of elements are cached in Level Buffer. We will show in our experimental result the detailed comparison and analysis. We omit the proof of THEOREM 4 for limited space.

In the case that the given query contains *P-C* or *F-S* edge, Algorithm LBHJ might produce root-to-leaf path solutions which do not match any solution of another root-to-leaf path. However, we will shown in experimental results that our method produce much less intermediate path solutions than naive method.

VI. Experimental Evaluation

A. *Experimental Setting*

Our experiments are implemented on a PC with 2.00 GHz Core 2 Duo processor, 1 G memory, 120 GB IDE hard disk, and Windows XP professional as the operation system.

Because TwigStack can guarantee that all elements are scanned only once and no redundant output when considering query which only contains *A-D* relationship, we extend it to the naïve method TwigStack+Join, or ***TSJ***. Both algorithms are implemented using Microsoft Visual C++ 6.0.

We use three datasets, XMark [17], DBLP [8] and TreeBank [15] for our experiments. XMark is a well known synthetic XML dataset which features a moderately complicated and fairly irregular schema, with several deeply recursive tags. DBLP is a highly regular dataset while

TreeBank is a highly irregular dataset. The main characteristics of these three datasets can be found in Table 1.

TABLE 1 STATISTICS OF XML DATASETS

Dataset	Size(M)	Nodes (Million)	Max Depth	Average Depth
DBLP	127	3.3	6	2.9
XMark	113	1.7	12	5.5
TreeBank	82	2.4	36	7.8

The queries used in our experiment are shown in Table 2. Among these queries, the first 7 queries, i.e. Q1-Q7, are queries without *F-S* axes, which we denote as the 1st group of queries, Q8-Q14 are queries with *A-D*, *P-C* and *F-S* axes, which we denote as the 2nd group of queries.

TABLE 2 QUERIES USED IN OUR EXPERIMENT

Query	Xpath Expression	Dataset
Q1	//people//person[.//name][.//age]	XMark
Q2	//listitem//parlist[.//bold]//text	XMark
Q3	//article[.//author]//title	DBLP
Q4	//book[.//author]//isbn	DBLP
Q5	//S//VP/PP[NP//VBN]//IN	TreeBank
Q6	//S[.//VP]/PP	TreeBank
Q7	//S[JJ]//NP	TreeBank
Q8	//title//sup[.//i]/following-sibling:sub	DBLP
Q9	//article//title//sup/following-sibling:sub	DBLP
Q10	//NP[.//NN/following-sibling:JJ]/PP//PRP_DOLLAR_	TreeBank
Q11	//NP/IN/following-sibling:PP	TreeBank
Q12	//NP[NN/following-sibling:JJ]//PP//PRP_DOLLAR_	TreeBank
Q13	//NP[NN/following-sibling:JJ//IN]//PP//PRP_DOLLAR_	TreeBank
Q14	//S//VP//NP//PP [following-sibling:NN]//IN//DT	TreeBank

We consider the following two performance metrics to compare the performance of these two algorithms: (1) **Number of the intermediate path solutions, which** reflects how a CTP processing algorithm can reduce the intermediate redundancy. (2) **Running time**, which reflects the CPU cost of algorithm.

B. Performance comparison and analysis

For queries with *F-S* axes, i.e. the 2nd group of queries, as shown in Table 3, LBHJ produces much less intermediate path solutions than TSJ. For example, consider Q14, the intermediate results of TSJ is 1023500, however, the number of final result is 24, which means large amount of redundant intermediate results are generated by the algorithm and they do not participate in any final answer. At the same time, we can see from this table that the number of intermediate results of LBHJ for Q14 is 1220, which is much less than 1023500. The reason lies in the fact that for each CTP query, it will be split into multiple twig pattern queries at the *F-S* edges, and each one is processed separately. This strategy will produce large amount of intermediate results which only satisfy one of the decomposed twig pattern, but not the whole CTP. In our method, however, we process the given CTP as a whole without decomposing it into several independent twig pattern queries and processing them one by one, thus the number of intermediate results of LBHJ is much less than that of TSJ.

For the same group of queries, as shown in Figure 7, also, we can see that LBHJ is much more efficient than TSJ. The reason is obvious, since large amount of intermediate path solutions are produced by TSJ, it needs more time for the second merge phase of the algorithm to decide which intermediate result is useful.

TABLE 3 INTERMEDIATE PATH NUMBER OF Q8-Q14

Query	TSJ Path	LBHJ Path	Reduction Percentage
Q8	737	65	91.18%
Q9	715	194	72.87%
Q10	161189	242	99.85%
Q11	62200	626	98.99%
Q12	64492	181	99.72%
Q13	74218	358	99.52%
Q14	1023500	1220	99.88%

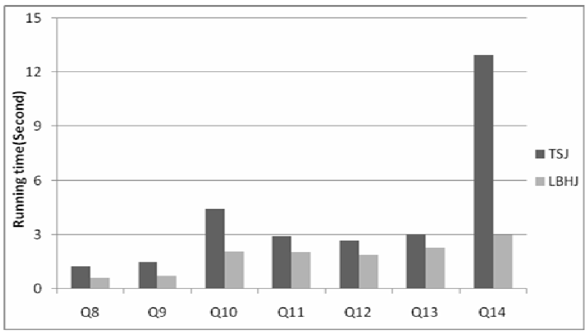


Fig. 7 The comparison of running time

For queries without *F-S* axes, i.e. the 1st group of queries, Figure 8 shows the experimental results of running time. We can see that LBHJ and TSJ have very similar performance because in these cases queries do not contain *F-S* relationship, and they produce same intermediate path solutions, since our method need some additional operation, we can see from Figure 8 that our method is little slower than TSJ, however, this is acceptable in practice compared with the huge benefits we got from processing CTP queries with *F-S* relationship.

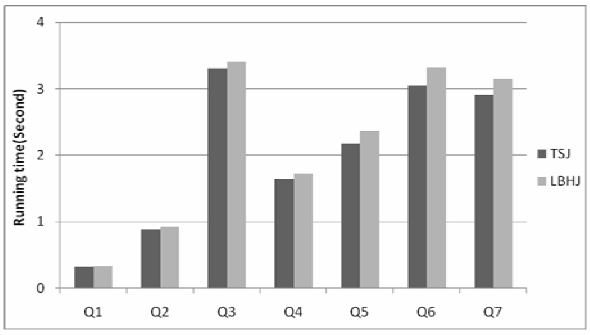


Fig. 8 The comparison of running time

As discussed in Section 5, in worst case, our method needs to cache all elements of the document. In practice, this worst case rarely happens. We show some experimental results about maximal buffer size for LBHJ in Figure 9, from which we can see that the buffer size is usually small enough to be cached in the main memory, this is because usually in parctice, the value of $Fanout_{do}$ is usually very small, and more important, our buffering strategy can largely reduce the number of buffered elements, thus only limited elements need to be cached in buffer at running time.

From the above experimental results and our analysis we know that when processing queries with *F-S* edges, LBHJ can work much more efficiently than TSJ. The reason lies in the fact that we process a CTP without decomposing it into several twig pattern and processing them separately. Thus our method produce much less intermediate path solutions, which will cause great performance improvement compared with TSJ. Even if no *F-S* edge appear in the query expression, our method still achieves similar performance compared to TSJ.

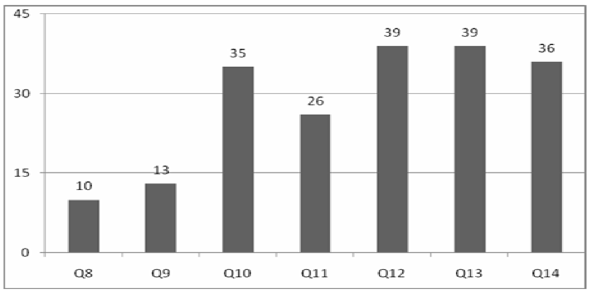


Fig. 9 Maximal buffer size at running time

VII. Related Work

Most XML query processing algorithms use a special positional representation to represent elements, like region encoding introduced by Zhang et al. [18] to XML query processing; alternatively, Tatarinov et al. [14] introduced Dewey ID labeling scheme to represent XML order in the relational data model..

Twig pattern is a core component of XML query languages to match data fragments in an XML document. MPMGJN [18] was first proposed for efficient binary structural join, Stack-Tree-Desc/Anc [1] improves the query performance of MPMGJN by using stack-based algorithm. Wu et al [16] further optimized the query performance by studying the problem of binary join order selection. Binary structure join methods like[18, 1, 16] suffer from large number of redundant intermediate results. To solve this problem, many approaches [4,7,9,5,10] were proposed to process a twig query holistically, and they avoid producing large size of useless intermediate results. Among them, TwigStack [4] was the first one proposed to process a twig pattern query in a holistic way. When considering query with only *A-D* relationship, TwigStack can guarantee that the CPU time and I/O optimal. Other methods [7, 9, 5, 10] made improvements to TwigStack from different aspects. TSGeneric+ [7] focused on holistic twig joins on all/partly indexed XML documents to skip many useless elements. TwigStackList [9] can reduce the intermediate path by attaching an element list to each query node to buffer necessary elements. Chen et al. proposed iTwigJoin [5] that exploits different data partition strategies to further boost the holism. And by using extended Dewey, TJFast [10] can improve query performance by accessing only leaf elements. Choi. et al [6] has proved that optimality evaluation of twig patterns with arbitrarily mixed *A-D* and *P-C* relationships is impossible. All these methods can be used to process CTP query, however, they cannot work efficiently since they do not process CTP in a holistic way.

All the approaches above only concerned unordered queries. Lu et al. [11] proposed a holistic algorithm, namely OrderedTJ, for ordered twig queries. It extends TwigStackList to handle ordered twig query. This algorithm is I/O optimal only when the *P-C* edge is the only or the first edge of one query node. However, there are many sorts of CTPs which contains ordered relationship can not be handled by this algorithm.

VIII. Conclusions

In this paper we addressed the problem of matching Complex Twig Pattern (CTP) which contains both containment and following-sibling relationships. We proposed a holistic join algorithm LBHJ based on an efficient buffering strategy for CTP query evaluation. Experimental results indicated that our Algorithm LBHJ can perform significantly better than the existing approach.

For the future work, we will continuously focus on the query evaluation of CTP containing following axis.

References

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

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. W3C Recommendation 23 January 2007.

[3] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu. XQuery 1.0: An XML Query. W3C Recommendation 23 January 2007.

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

[5] T. Chen, J. Lu, and T.W. Ling. On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques. In Proceedings of SIGMOD 2005, pages 455-466, 2005.

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

[7] 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, 2003.

[8] M. Ley. DBLP database web site. http://www.informatik.uni-trier.de/~ley/db .

[9] J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In Proceedings of CIKM 2004, pages 533–542, 2004.

[10] J. Lu, T.W. Ling, C.Y. Chan, and T. Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In Proceedings of VLDB 2005, pages 193-204, 2005.

[11] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Efficient Processing of Ordered XML Twig Pattern. In Proceedings of DEXA 2005, pages 300-309, 2005.

[12] Olteanu D. Forward node-selecting queries over trees. ACM Trans. Database Syst. 32(1): 3 (2007).

[13] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In Proceedings of EDBT Workshops 2002，pages 109-127, 2002.

[14] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasun daram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In Proceedings of SIGMOD 2002, pages 204-215, 2002.

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

[16] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In Proceedings of ICDE 2003, pages 443-454, 2003.

[17] XMark. http://monetdb.cwi.nl/xml.

[18] 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 SIGMOD 2001, pages 425-436, 2001.