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 cannot 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 (a1, c1), (a2, c2), …, (a1, c5), (a1, d1), (a1, d2), (a1, d3), 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. (a1, c1, d1), (a1, c1, d2), (a1, c1, d3).

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 naïve 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, $\Sigma$ 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](image)

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 tuple (Start, End, Level) based on its left to right deep-first traversing the document. An element $u$ is an ancestor of another element $v$ if $u$.Start $<$ $v$.Start $< u$.End. $u$ is the parent of $v$ if $u$.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 if $u$.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 $u$.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, ‘\rightarrow’ denotes A-D relationship, and ‘\rightarrow’ 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](image)

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 in S
3. Output intermediate results of Q using TwigStack(Q)
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 decompose 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 path solutions, i.e. (a1, b1, d1), (a1, f1), (a1, f2), (a1, f3). The second set consists of three path solutions, i.e. (a1, g1), (a1, g2). In line 4, all these path solutions are merged together to get the final answers. In fact, however, only (a1, b1, d1), (a1, f2), and (a1, g2) 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 Cc points to elements with tag C, Cd points to elements with tag D and Cc moves according to the position of Cd, if Cd points to d2, Cc need to move from c1 to c2 to join with d2. After Cd moves to d3, Cc needs to backtrack from c3 to c1 so as to join with d3, 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 p1, we use C\(_p\) to denote the current processed element with tag F, and further we assume p1 << C\(_p\). Then we have to consider three cases for C\(_p\):

a. C\(_p\) equals to f1 or f2, which is the following sibling of p1, i.e. p1.Start < f1.Start and p1.ParEnd = f1.ParEnd (same condition holds for f3).

b. C\(_p\) equals to f2, which is an element in R(p1), and not a following sibling of p1. Note that although f2 is not a following sibling of p1, maybe there is another element that is a following sibling of p1, e.g. f3 in Figure 4 (b), so p1 should be cached for further processing. In this case p1.Level << f2.Level, p1.Start < f2.Start < parent(p1).ParEnd.

c. C\(_p\) equals to f3 and is not in R(p1), which means that p1 does not match f3, so we can safely discard p1 and process the next element with tag P. In this case p1.ParEnd < f3.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 y 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, f2 and g2 have the same parent c2 and f2 << g2. Assume a1.Level = 1. According to the documental order, for any element x that satisfies f2 << x << g2, then x ∈ {e1, c3, d3}, so x.Level ≤ 3, and f2.Level = g2.Level = 3.

**THEOREM 1:** Consider the CTP query //P//following-sibling::F. Assume elements p1 and p2 has tag P, element f has tag F. All the three elements satisfy that p1 << p2, f << p1, p1.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 $p_1 \prec f$ and $p_1.Level = f.Level$, according to Lemma 1, we have $p_2.Level \geq p_1.Level$, 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 $d_5$. Since $c_2 \prec c_4 \prec d_3$ 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 $c_3 \prec c_4 \prec d_2$ 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 $w$ ($u \prec v \prec w$), $u$.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 Lemma 1, $v$ 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 $f_1 \prec f_2 \prec g_2$, $f_1$.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 $u \prec v$, 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](Image)

**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 $d_1 \prec c_1$, $C_D$ is moved to $d_1$ 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 $d_3$ 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 $c_3$ and $c_4$ 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 $e_{parent(q)}$ in $LB_{parent(q)}$ 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) $\text{isLeaf}(q) \land C_q \neq \text{NULL}$, or
2) for each $q' \in \text{children}(q)$
   (i) $q'/q' \land \text{hasPSE}(q') \land C_{q'/q'}^2 \land C_{q'/q'}^2, or$
   (ii) $q'q' \land \text{hasPSE}(q') \land (C_{q'/q'} \land C_{q'/q'})$

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 $LB_{parent(q)}$ that are useless according to $C_q$, which is further classified into two cases: (1)
parent(q)/q, then all elements that are not ancestor of \(C_p\) will be popped from \(LB_q\). (2) parent(q) \(\rightarrow q\), then all elements that have larger Level value than \(C_q\) 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 \(LB\) (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(), \(\text{nearestAnc}(LB, C_q)\) is used to get the lowest ancestor of \(C_q\) from \(LB\), and \(\text{nearestEle}(LB, C_q)\) 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 (!\(\text{end}(Q)\)) do
2: \(q = \text{getNext}(Q)\)
3: if not isRoot(q) then
4: \(\text{cleanLB}(LB, C_q)\)
5: if isRoot(q) or \(\text{hasMatchedEle}(LB, C_q)\) then
6: \(\text{cleanLB}(LB, C_q)\)
7: \(\text{Push}(LB, C_q)\)
8: if isLeaf(q) then
9: \(\text{outputPathSolutionsWithBlocking}(C_q)\)
10: \(\text{Advance}(C_q)\)
11: end while
12: \(\text{MergeAllPathSolutions}()\);

Procedure cleanLB(LB, C_q)
1: if q/p then
2: Pop all elements that are not ancestor of \(C_p\) from \(LB\)
3: if q then
4: Pop all elements that have larger Level value than \(C_q\) or elements that have same Level value but not same parent with \(C_q\) from \(LB\)

Function hasMatchedEle(LB, C_q)
1: if \(\exists e \in LB / q/p \land e/\) then return TRUE
2: if \(\exists e \in LB / q/p \land e/\) then return TRUE
3: return FALSE

Procedure Push(LB, C_q)
1: \(e = \text{nearestAnc}(LB, C_q)\)
2: if \(e = \text{NULL}\) then \(C_p\text{pSelfA} = e\)
3: else \(C_p\text{pSelfA} = \text{NULL}\)
4: \(e = \text{nearestEle}(LB, C_q)\)
5: if isRoot(q) then \(C_p\text{StrucA} = \text{NULL}\)
6: else \(C_p\text{StrucA} = e\)

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 elements running. 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 easy 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 \(n_{\text{min}}\) and \(n_{\text{max}}\) which have the minimal and maximal Start value from all children that has A-D relationship with \(q\). In line 7, we find \(n_{\text{min}}\) 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 \(C_{\text{min}}\).Start. If \(C_q\).Start is larger than \(C_{\text{min}}\).Start, \(n_{\text{min}}\) is returned in line 10. In line 11, if \(C_q\).Start is larger than \(C_{\text{min}}\).Start, \(n_{\text{min}}\) 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 \(p \in \text{children}(q)\) do
3: \(p' = \text{getNext}(p)\)
4: if \(p' \neq p\) then return \(p'\)
5: \(n_{\text{min}} = \text{minarg}(C_p\text{.Start} / q/p)\)
6: \(n_{\text{max}} = \text{maxarg}(C_p\text{.Start} / q/p)\)
7: \(\{r = \text{minarg}(C_p\text{.Start} / q/p)\}
8: while \((C_p\text{.End} < C_{\text{min}}\text{.Start})\) do
9: \(\text{Advance}(C_p)\)
10: if \(C_p\text{.Start} \geq C_{\text{min}}\text{.Start}\) then return \(n_{\text{min}}\)
11: if \(C_p\text{.Start} > C_{\text{min}}\text{.Start}\) then return \(r_{\text{min}}\)
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 \(C_p\text{.Start} = a_1\), 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 \(C_p\text{.Start} = b_1\), since \(b_1\) has a matched element in \(LB_B\), 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 \(C_p\text{.Start} = c_1, c_2, c_3, c_4, c_5\), 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 \(C_p\text{.Start} = d_2\), as shown in Figure 6 (i), \(d_2\) will be pushed into \(LB_D\), and \(p_p\text{.StrucA}\) points to c5. The tenth call of getNext(A) returns C with \(C_p\text{.Start} = c_6\), since the parent of \(c_6\) is not equal to that of \(c_6\), c5 is popped from \(LB_D\), 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 \(C_p\text{.Start} = d_3\), since C and D have F-S relationship and the Level value of \(d_1\) is less than that of \(c_6, c_6\).
is popped from LB2, then d1 is pushed into LB2, the current status is shown in Figure 6 (k). The remaining 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 output when an element of leaf node is pushed into a LB, and getNext(A) is similar to the above description and we omit for simplicity.

**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 $O(Input\ Data\ Size\ +\ Inter\ Result\ Size\ +\ Output\ Result\ Size)$.

The proof is obvious in worst case. Level buffer will cache all 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>
<thead>
<tr>
<th>Dataset</th>
<th>Size (M)</th>
<th>Nodes (Million)</th>
<th>Max Depth</th>
<th>Average Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>127</td>
<td>3.3</td>
<td>6</td>
<td>2.9</td>
</tr>
<tr>
<td>XMark</td>
<td>113</td>
<td>1.7</td>
<td>12</td>
<td>5.5</td>
</tr>
<tr>
<td>TreeBank</td>
<td>82</td>
<td>2.4</td>
<td>36</td>
<td>7.8</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Query</th>
<th>Xpath Expression</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>//people//person[@name [@/age]]</td>
<td>XMark</td>
</tr>
<tr>
<td>Q2</td>
<td>//listitem//parlist[@/bold]//text</td>
<td>XMark</td>
</tr>
<tr>
<td>Q3</td>
<td>//article[@/author]//title</td>
<td>DBLP</td>
</tr>
<tr>
<td>Q4</td>
<td>//book[@/author]//isbn</td>
<td>DBLP</td>
</tr>
<tr>
<td>Q5</td>
<td>//S//VP//PP[NP//VBN]//IN</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q6</td>
<td>//S//VP//PP</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q7</td>
<td>//S//JJ//NP</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q8</td>
<td>//title//sup[@/i]/following-sibling:sub</td>
<td>DBLP</td>
</tr>
<tr>
<td>Q9</td>
<td>//article/title//sup/following-sibling:sub</td>
<td>DBLP</td>
</tr>
<tr>
<td>Q10</td>
<td>//NP//NN/following-sibling:JJ//PP//PR P DOLLAR</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q11</td>
<td>//NP//IN/following-sibling:PP</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q12</td>
<td>//NP//NN/following-sibling:JJ//PP//PR P DOLLAR</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q13</td>
<td>//NP//NN/following-sibling:JJ//IN//PP//PRP DOLLAR</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q14</td>
<td>//S//NP//NP//PP [following-sibling:NN]//IN//DT</td>
<td>TreeBank</td>
</tr>
</tbody>
</table>

Table 2 Queries Used in Our Experiment

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 result 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.

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.
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 practice, the value of $Fanout_{db}$ 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 $F$-3 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 $F$-3 edge appear in the query expression, our method still achieves similar performance compared to TSJ.

When considering query with only $A$-$D$ ordered edge, LBHJ can further boost the holism. And by using extended Dewey, TFast [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