Efficient Processing of Complex Twig Pattern Matching

Jinqing Zhu, Wei Wang, Xiaofeng Meng

Renmin University of China, Beijing, China
{zhujinqing, dawei, xf meng}@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 relationships 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.

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).

OrderTJ [11] was proposed to handle ordered query, but for the query /A/C/follow-sibling:D, it needs to translate the query to /A/*[C]/follow-sibling:D, where * denotes any tag, 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/follow-sibling::B, and (2) query patterns with mixed order and unordered relationships, e.g. /A/B/C/follow-sibling:D.

In this paper, we propose a novel method named LBHJ that can process CTP with follow-sibling relationship appearing at any place in the CTP. And LBHJ processes CTPs in a holistic way, so that its performance is much efficient.

Our contributions can be summarized as follows:

• Give an analysis of the following-sibling relationship, and present some conclusions that can be used to avoid redundant operations when evaluating a CTP.
• Propose a novel data structure, Level Buffer, to cache temporal results, and propose a new holistic join algorithm LBHJ, which can find answers of the given CTP efficiently in a holistic way using Level Buffer.

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.

Fig. 1 A sample XML document

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.
Most existing XML query processing algorithms rely on a positional representation of elements, e.g., region encoding scheme [17], 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 if and only 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.

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. In this work, we extend region encoding scheme by adding one additional field ParEnd which 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. [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-CÆ, where ‘Æ’ denotes P-C relationship, ‘/’ denotes A-D relationship, and ‘Æ’ 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.

Matching a CTP against an XML database is to find all occurrences of the pattern in the database. Consider the CTP //A[B[D]]/FÆG and the XML document in Figure 1. One match of it is (a1, b1, d1, f2, g2).

III. NAIVE SOLUTION

In this section, we present a naive solution 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 Qi in S
3: Output intermediate results of Qi using TwigStack(Qi)
4: Merge all intermediate results to get final answers

EXAMPLE 1: Consider //A[B[D]]/FÆG and the XML document in Figure 1. Algorithm 1 decompose it into two twig patterns, i.e., //A[B[D]]/FÆF and //A/G. After processing each twig pattern separately using TwigStack algorithm, two sets of intermediate results will be produced. In line 4, the intermediate path solutions are merged together to get the final answers.

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. Buffering Guidelines:

For the CTP query //CÆD, assume c1, c2 are elements with C tag, and d1 is an element with D tag in one XML document. And c1.Start < c2.Start, c1.Start < d1.Start. We know that even if d1 is not a follow sibling of c1, d1 may be a follow sibling of c2, so when processing c1, d1 need be cached. So in this section we give a guideline for caching.

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.

THEOREM 1: Consider the CTP query //PÆF. Assume elements p1 and p2 has tag P, element f has tag F. if p1 << p2 and p1.Level > p2.Level, then, p1 cannot be a preceding sibling of f, and p1.ParEnd != f.ParEnd.

Proof: Assume that p1 and f have F-S relationship, then p1 << f and p1.Level > f.Level, according to Lemma 1, we have p2.Level >= p1.Level, which contradicts the given condition p1.Level > p2.Level.

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, according Lemma 1, v must be a descendant of the parent of u, since u.Level = v.Level, then u.ParEnd = v.ParEnd. This conflicts with the condition u.ParEnd != v.ParEnd, so u cannot be a preceding sibling of w.

Based on the above analysis, we introduce a new data structure, Level Buffer (LB). LB is a chain of linked list, among which each list contains elements with the same Level value. Moreover, we have 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

EXAMPLE 2: Consider the XML document in Figure 2 and the CTP query //CÆD. Initially, cursors Cc and Cd point to c1 and d1, respectively. Since d1 << c1, Cd is moved to d2 and c1 is pushed into LB, then Cc is moved to c2, then c2 and c1 are pushed into LB since their Level value are larger than that of c1 and they appear before d1 in documental order. After that, Cc points to c3, then we safely discard c2 and c1 since the Level value of c4 is less than that of c2 and c3. After c4 and c5 are pushed into LB, the status of LB is shown in Figure 2.

1 for element u and v, we say u <= v if u.Start < v.Start
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 \( T_q \). \( C_q \) points to elements in \( T_q \), especially, \( C_q \) is NULL if all elements in \( T_q \) are processed, and \( C_q \) is also used to denote the element it points to. At the beginning, all cursors point to the first elements of 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 query root or query leaf. The function children\((q)\) is used to return all the child query nodes of \( q \) and parent\((q)\) is used to return the parent node of \( q \).

What should be noticed is that each element \( e_p \) in LB\(_p\) has two pointers, one is \( e_p\_pSelfA\), which points to the nearest ancestor element that has the same tag in LB\(_p\), the other is \( e_p\_pStrucA\), 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, 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 \( q’\)’s children\((q)\)
   (i) \( q’/q’ \) hasPSE\((q’')\) \&\& \( C_{q’’}/C_q \), or
   (ii) \( q’/q’ \) hasPSE\((q’')\) \&\& \( C_{q’’} < C_q \)

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

Function hasMatchedEle(LB\(_p, C_p)\)
1: if \( \exists e \in LB\((q/p \&\& e/C_q)\) then return TRUE
2: if \( \exists e \in LB\((q/p \&\& e/C_q)\) then return TRUE
3: return FLASE

Procedure Push(LB\(_p, C_p)\)
1: \( e = \) nearestAnc(LB\(_p, C_p)\)
2: if \( e/\) is NULL then \( C_p\_pSelfA = e\)
3: else \( C_p\_pSelfA = \) NULL
4: \( e = \) nearestEle(LB\(_parent(e), C_p)\)
5: if isRoot\((q)\) then \( C_p\_pStrucA = \) NULL
6: else \( C_p\_pStrucA = e\)

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 \( C_q \) will be popped from LB\(_q\); (2) parent\((q)/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\(_q\) (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\(_p, C_p)\) is used to get the lowest ancestor of \( C_q \) from LB\(_p\), 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 \( q \).

What should be noticed in LBHJ is that the conclusions we get 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 easy to understand that our method will achieve similar performance for CTP with only $A-D$ edges.

**Algorithm 3 getNext(q)**

1. if isLeaf(q) = TRUE then return q
2. for $p \in$ children(q) do
3. $p' = \text{getNext}(p)$
4. if $p' \neq p$ then return $p'$
5. $n_{\text{min}} = \min\arg\{C_p.\text{Start} | q/p\}$
6. $n_{\text{max}} = \max\arg\{C_p.\text{Start} | q/p\}$
7. $r_{\text{min}} = \min\arg\{C_p.\text{Start} | q \rightarrow p\}$
8. while ($C_q.\text{End} < C_{\text{minStart}}.\text{Start}$) do
9. Advance($C_q$)
10. if $C_q.\text{Start} > C_{\text{minStart}}.\text{Start}$ then return $n_{\text{min}}$
11. if $C_q.\text{Start} > C_{\text{minStart}}.\text{Start}$ then return $r_{\text{min}}$
12. return q

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 minimum and maximal Start value from all children that has $A-D$ relationship with $q$. In line 7, we find $r_{\text{min}}$ 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 $C_q.\text{End}$ is not less than $C_{\text{minStart}}.\text{Start}$. If $C_q.\text{Start}$ is larger than $C_{\text{minStart}}.\text{Start}$, $n_{\text{min}}$ is returned in line 10. In line 11, if $C_q.\text{Start}$ is larger than $C_{\text{minStart}}.\text{Start}$ then 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.

**Example 3:** Consider //A[[B]]//C-D 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 3. 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 $C_B$ pointing to $a_1$, since A is root node, $a_1$ is pushed into LB$_B$, 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$_B$, i.e. $a_1$, $b_1$ is pushed into LB$_B$, the LB status is shown as Figure 3 (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 3 (d-h), each element in Level Buffer has two pointers, one is pSelfA, shown as blue arrows, and the other is pStrucA, shown as red arrows. The next call of getNext(A) returns D with $C_D$ pointing to $d_3$, as shown in Figure 3 (i), $d_3$ will be pushed into LB$_D$, and $D$.pStrucA points to $c_5$. The remainder four 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 $a_1$, $b_1$, (a1, c1, d1), (a1, c1, d3) and (a1, c1, d1) in the second phase of LBHJ, all the four path solutions are merged together to get the final answers, they are $a_1$, $b_1$, $c_1$, $d_1$, $a_1$, $b_1$, $c_1$, $d_3$ and $a_1$, $b_1$, $c_1$, $d_3$.

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$_{\text{parent}(q)}$ (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 $e_{\text{parent}(q)}$ in LB$_{\text{parent}(q)}$ that satisfies the structural relationship of $<\text{parent}(q), q>$ 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 modify two pointers, i.e. $e_{q}.\text{pSelfA}$ and $e_{q}.\text{pStrucA}$, the former points to its nearest ancestor element in LB$_q$, maintaining the $A-D$ relationship among elements in the same LB, the other points to the nearest element $e_{\text{parent}(q)}$ in LB$_{\text{parent}(q)}$ which satisfies the structural constraint of parent($q$) and $q$ with $C_q$. If $q$ is a leaf node, all path solutions related with $C_q$ is produced in line 10 of Algorithm 2. 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

![](image.png)
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 **THEOREM 4** is obvious in worst case. We omit the proof for limited space.

### 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.

We extend TwigStack to the naïve method TwigStack+Join, or **TSJ**. Both algorithms are implemented using Microsoft Visual C++ 6.0.

We use XMark [16], DBLP [8] and TreeBank [14] 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**

<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.

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, 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 only 24, which means large amount of useless intermediate results are generated by the algorithm. At the same time, we can see that the number of intermediate results of LBHJ for Q14 is 1220, which is much less than 1023500. The reason lies in the fact TSJ splits every CTP query 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. In our method, however, we process the given CTP as a whole.

For the same group of queries, as shown in Figure 4, also, we can see that LBHJ is much more efficient than TSJ. Because, having produced large amount of intermediate path solutions, TSJ needs more time for the second merge phase to decide which one is useful.

**TABLE 2 QUERIES USED IN OUR EXPERIMENT**

<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>//VP/PP[NP//VBN]//IN</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q6</td>
<td>//VP//VP/PP</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q7</td>
<td>//S[J]/NP</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q8</td>
<td>//title//sup//j/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:J]/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/IN/following-sibling:J]/PP//PR P_DOLLAR_</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q13</td>
<td>//NP/IN//following-sibling:J]/IN//PP//PR P_DOLLAR_</td>
<td>TreeBank</td>
</tr>
<tr>
<td>Q14</td>
<td>//S//VP/PP/PPP [following-sibling:NN]//IN//DT</td>
<td>TreeBank</td>
</tr>
</tbody>
</table>

**TABLE 3 INTERMEDIATE PATH NUMBER OF Q8-Q14**

<table>
<thead>
<tr>
<th>Query</th>
<th>TSJ Path</th>
<th>LBHJ Path</th>
<th>Reduction Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q8</td>
<td>737</td>
<td>65</td>
<td>91.18%</td>
</tr>
<tr>
<td>Q9</td>
<td>715</td>
<td>194</td>
<td>72.87%</td>
</tr>
<tr>
<td>Q10</td>
<td>161189</td>
<td>242</td>
<td>99.85%</td>
</tr>
<tr>
<td>Q11</td>
<td>62200</td>
<td>626</td>
<td>98.99%</td>
</tr>
<tr>
<td>Q12</td>
<td>64492</td>
<td>181</td>
<td>99.72%</td>
</tr>
<tr>
<td>Q13</td>
<td>74218</td>
<td>358</td>
<td>99.52%</td>
</tr>
<tr>
<td>Q14</td>
<td>1023500</td>
<td>1220</td>
<td>99.88%</td>
</tr>
</tbody>
</table>

For queries without \(F-S\) axes, i.e. the 1st group of queries, Figure 4 shows the experimental results of running time. We can see that LBHJ and TSJ have very similar performance because they produce same intermediate path solutions for these queries. Since our method need some additional operation, we can see that our method is little slower. However, this is acceptable in practice, compared with the huge benefits we got from processing 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 5, from which we can see that the buffer size is usually small enough to be cached in the main memory, this is because, in practice, the value of \(Fanout_{doc}\) 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 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. Even if no F-S edge appear in the query expression, our method still achieves similar performance compared to TSJ.

To solve this problem, many approaches [4,7,9,5,10] were proposed to process a twig pattern 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, 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, 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.

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