Unordered Tree Matching and Strict Unordered Tree Matching: the Evaluation of Tree Pattern Queries

Yangjun Chen, Donovan Cooke

Dept. Applied Computer Science, University of Winnipeg
515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9
y.chen@uwinnipeg.ca

Abstract — In this paper, we consider two kinds of unordered tree matchings for evaluating tree pattern queries in XML databases. For the first kind of unordered tree matching, we propose a new algorithm, which runs in $O(|D|Q)$ time, where $Q$ is a tree pattern and $D$ is a largest data stream associated with a node of $Q$. It can also be adapted to an indexing environment with XB-trees being used to speed up disk access. Experiments have been conducted, showing that the new algorithm is promising. For the second of tree matching, the so-called strict unordered tree matching, we show that the problem is NP-complete by a reduction from the satisfiability problem.

Key words: Tree mapping, twig pattern, XML databases, query evaluation, tree encoding

I. INTRODUCTION

In XML [33, 34], data are represented as a tree; associated with each node of the tree is an element name tag from a finite alphabet $\Sigma$. The children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element.

To abstract from existing query languages for XML (e.g. XPath [15], XQuery [34], XML-QL [14], and Quilt [6, 7]), we express queries as tree patterns, where nodes are labeled with symbols from $\Sigma \cup \{\ast\}$ ($\ast$ is a wildcard, matching any node name) and string values, and edges are parent-child or ancestor-descendant relationships. As an example, consider the query tree shown in Fig. 1.

Fig. 1. A query tree

This query asks for any node of name $b$ (node 3) that is a child of some node of name $a$ (node 1). In addition, the node of name $b$ (node 3) is the parent of some nodes of name $c$ and $e$ (node 6 and 7, respectively), and the node of name $e$ itself is an ancestor of some node of name $d$ (node 8). The node of name $b$ (node 2) should also be the ancestor of a node of name $f$ (node 5). The query corresponds to the following XPath expression: 

$$a[b[c \text{ and } /f]]/b[c \text{ and } e/d]$$

In this figure, there are two kinds of edges: child edges (/edges for short) for parent-child relationships, and descendant edges ///</edges for short) for ancestor-descendant relationships.

A /-edge from node $v$ to node $u$ is denoted by $v \to u$ in the text, and represented by a single arc; $u$ is called a /-child of $v$. A ///</edge is denoted $v \Rightarrow u$ in the text, and represented by a double arc; $u$ is called a ///</child of $v$.

In any DAG (directed acyclic graph), a node $u$ is said to be a descendant of a node $v$ if there exists a path (sequence of edges) from $v$ to $u$. In the case of tree patterns, this path could consist of any sequence of /-edges and/or ///</edges. We also use $\text{label}(v)$ to represent the symbol ($\in \Sigma \cup \{\ast\}$) or the string associated with $v$. Based on these concepts, the tree embedding can be defined as follows.

Definition 1. An embedding of a tree pattern $Q$ into an XML document $T$ is a mapping $f: Q \to T$, from the nodes of $Q$ to the nodes of $T$, which satisfies the following conditions:

(i) Preserves node label: For each $u \in Q$, $\text{label}(u) = \text{label}(f(u))$.
(ii) Preserves parent-child/ancestor-descendant relationships: If $u \to v$ in $Q$, then $f(v)$ is a child of $f(u)$ in $T$, if $u \Rightarrow v$ in $Q$, then $f(v)$ is a descendant of $f(u)$ in $T$.

If there exists a mapping from $Q$ into $T$, we say, $Q$ can be embedded into $T$, or say, $T$ contains $Q$.

This definition allows a path to match a tree as illustrated in Fig. 2.

Fig. 2. A tree matches a path

It even allows to map several nodes with the same tag name in a query to the same node in a database. In fact, it is a kind of unordered tree matching, by which the order of siblings is not significant. Almost all the existing strategies for evaluating tree pattern queries are designed according to this definition [5, 9, 10, 11, 14, 22, 24, 26, 27, 28, 29, 34, 35]. Such kind of tree matchings can be solved in polynomial time. However, if we require that each query node in $Q$ maps to a different document node in $T$ and no siblings map to those nodes which are related by ancestor/descendant or parent/child relationships, the problem becomes very difficult. We refer to it as strict unordered tree matching.

In this paper, we first present a new algorithm for evaluating tree pattern queries according to Definition 1, which runs in $O(|D||Q|)$ time and $O(|D||Q|)$ space, and can be adapted to an indexing environment with XB-trees being used, where $D$ is a...
largest data stream associated with a node $q$ of $Q$. Then, we show that the strict unordered tree matching is $\textit{NP}$-complete.

The remainder of the paper is organized as follows. In Section 2, we restate a tree encoding [36], which facilitates the recognition of different relationships among the nodes of a tree. In Section 3, we discuss our algorithm for the unordered tree matching according to Definition 1. In Section 4, we show the $\textit{NP}$-completeness of the strict unordered tree matching. Section 6 is devoted to the implementation and experiments. Finally, a short conclusion is set forth in Section 6.

II. TREE ENCODING

In [36], an interesting tree encoding method was discussed, which can be used to identify different relationships among the nodes of a tree.

Let $T$ be a document tree. We associate each node $v$ in $T$ with a quadruple $(\text{DocId}, \text{LeftPos}, \text{RightPos}, \text{LevelNum})$, denoted as $\alpha(v)$, where $\text{DocId}$ is the document identifier; $\text{LeftPos}$ and $\text{RightPos}$ are generated by counting word numbers from the beginning of the document until the start and end of $v$, respectively; and $\text{LevelNum}$ is the nesting depth of $v$ in the document. (See Fig. 3 for illustration.) By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined [36]:

(i) ancestor-descendant: a node $v_1$ associated with $(d_1, l_1, r_1, l_1)$ is an ancestor of another node $v_2$ with $(d_2, l_2, r_2, l_2)$ iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$.

(ii) parent-child: a node $v_1$ associated with $(d_1, l_1, r_1, l_1)$ is the parent of another node $v_2$ with $(d_2, l_2, r_2, l_2)$ iff $d_1 = d_2$, $l_1 < l_2$, $r_1 > r_2$, and $l_2 = l_1 + 1$.

(iii) from left to right: a node $v_1$ associated with $(d_1, l_1, r_1, l_1)$ is to the left of another node $v_2$ with $(d_2, l_2, r_2, l_2)$ iff $d_1 = d_2$, $r_1 < l_2$.

As with TwigStack [5], each node $q$ in a tree pattern (or say, a query tree) $Q$ is associated with a data stream $B(q)$, which contains the positional representations (quadruples) of the database nodes $v$ that match $q$ (i.e., $\text{label}(v) = \text{label}(q)$). All the quadruples in a data stream are sorted by their (DocID, LeftPos) values. For example, in Fig. 4, we show a query tree containing 5 nodes and 4 edges and each node is associated with a list of matching nodes of the document tree shown in Fig. 3, sorted according to their (DocID, LeftPos) values. For simplicity, we use the node names in a list, instead of the node’s quadruples.

Therefore, if each time we choose a node with the least LeftPos from data streams, $T$ is in fact traversed in preorder (top-down). However, our algorithm needs to visit the tree nodes in postorder (bottom-up). For this purpose, we maintain a global stack $S$ to keep all the nodes on a document path until we meet a node $v$ that is not a descendant of $S$.top (see line 3). Then, we pop up all those nodes that are not $v$’s ancestor, and push $v$ into $S$ (see lines 4 - 5). The output of the algorithm is a set of data streams $L(q_i)$’s with each being sorted by (DocID, RightPos). But we notice that the popped nodes themselves are in postorder (see line 3). So we can handle the nodes in this order without explicitly generating $L(q_i)$’s. However, in the following discussion, we assume that all $L(q_i)$’s are completely generated for ease of explanation. We also note that the data streams associated with different nodes in $Q$ may be the same. So we use $q$ to represent the set of such query nodes and denote by $L(q)$ the data stream shared by them. For example, in $Q$ shown in Fig. 4, $L((q_2, q_3)) = \{v_4, v_2, v_8\}$. Without loss of generality, assume that the query nodes in $q$ are sort-
ed by their RightPos values.

We will also use \( L(Q) = \{ L(q_1), \ldots, L(q_i) \} \) to represent all the data streams with respect to \( Q \), where each \( q_i \) \((i = 1, \ldots, l)\) is a set of sorted query nodes that share a common data stream.

We first observe that iterating through \( L(q_1), \ldots, L(q_i) \) corresponds to a navigation of \( T \) in postorder. However, in this process, any node \( v \) in \( T \), which does not match any \( q \in Q \) (i.e., \( \text{label}(v) \neq \text{label}(q) \)), is not accessed. So only a subtree \( T' \) of \( T \) is navigated. If we are able to construct \( T' \) explicitly in this process, we will get a tree structure with each node \( v \) associated with a query node stream \( S(v) \), as illustrated in Fig. 6. For each \( q \in S(v) \), we have \( \text{label}(v) = \text{label}(q) \).

![Fig. 6. Illustration for generating \( QS \)'s](image_url)

If we check, before a \( q \) is inserted into the corresponding \( S(v) \), whether \( Q[q] \) (the subtree rooted at \( q \)) can be embedded into \( T'[v] \), we get in fact an algorithm for tree pattern matching. The challenge is how to conduct such a checking efficiently.

For this purpose, we associate each \( q \) in \( Q \) with a variable, denoted \( \chi(q) \). During the process, \( \chi(q) \) will be dynamically assigned a series of values \( a_0, a_1, \ldots, a_m \) for some \( m \) in sequence, where \( a_0 = \phi \) and \( a_i's \) \((i = 1, \ldots, m)\) are different nodes of \( T' \). Initially, \( \chi(q) \) is set to \( a_0 = \phi \). \( \chi(q) \) will be changed from \( a_{i-1} \) to \( a_i = v \) \((i = 1, \ldots, m)\) when the following conditions are satisfied.

i) \( v \) is the node currently encountered.

ii) \( q \) appears in \( S(u) \) for some child node \( u \) of \( v \).

iii) \( q \) is a /-child, or

\( q \) is a \( l \)-child, and \( u \) is a \( l \)-child with \( \text{label}(u) = \text{label}(q) \).

Then, each time before we insert \( q \) into \( S(v) \), we will do the following checking:

1. Let \( q_1, \ldots, q_k \) be the child nodes of \( q \).

2. If for each \( q_i \) \((i = 1, \ldots, k)\), \( \chi(q_i) \) is equal to \( v \) and \( \text{label}(v) = \text{label}(q) \), insert \( q \) into \( S(v) \).

Since \( T' \) is constructed in a bottom-up way, the above checking guarantees that for any \( q \in S(v) \), \( T'[v] \) contains \( Q[q] \).

In terms of the above discussion, we give our algorithm for evaluating tree pattern queries. The algorithm mainly consists of a main procedure and a subprocedure. The task of the main procedure is to construct \( T' \) while the subprocedure is invoked to check tree embedding. In the main procedure, each node that is created for a quadruple \( v \) from \( L(q) \) is associated with two links, denoted respectively \( \text{left-sibling}(v) \) and \( \text{parent}(v) \), to mainly the tree structure of \( T' \) as follows:

1. Identify a data stream \( L(q) \) with the first element being the minimal RightPos value. Choose the first element \( v \) of \( L(q) \).

2. Remove \( v \) from \( L(q) \).

3. Generate a node for \( v \).

4. If \( v \) is not the first node, we do the following:

   - Let \( v' \) be the node created just before \( v \). If \( v' \) is not a child (descendant) of \( v \), create a link from \( v \) to \( v' \), called a \( \text{left-sibling} \) link and denoted as \( \text{left-sibling}(v) = v' \).

   - If \( v' \) is a child (descendant) of \( v \), we will first create a link from \( v \) to \( v' \), called a \( \text{parent} \) link and denoted as \( \text{parent}(v') = v \). Then, we will go along the left-sibling chain starting from \( v' \) until we meet a node \( v'' \) which is not a child ( descendant) of \( v \). For each encountered node \( u \) except \( v'' \), set \( \text{parent}(u) \leftarrow v \). Finally, set \( \text{left-sibling}(v) \leftarrow v'' \).

Fig. 7 is a pictorial illustration of this process.

In Fig. 7(a), we show the navigation along a left-sibling chain starting from \( v' \) when we find that \( v' \) is a child ( descendant) of \( v \). This process stops whenever we meet \( v'' \), a node that is not a child ( descendant) of \( v \). Fig. 8(b) shows that the left-sibling link of \( v \) is set to \( v'' \), which is previously pointed to by the left-sibling link of \( v \)'s left-most child.

![Fig. 7. Illustration for the construction of a matching subtree](image_url)

In Fig. 8, we give the main algorithm, by which a quadruple is removed in turn from the data streams \( L(q) \)'s and a node \( v \) for it is generated and inserted into \( T' \).

In addition, two data structures are used:

- \( D_{\text{root}} \) - a subset of document nodes \( v \) such that \( Q \) can be embedded in \( T'[v] \).

- \( D_{\text{output}} \) - a subset of document nodes \( v \) such that \( Q[q_{\text{output}}] \) can be embedded in \( T'[v] \), where \( q_{\text{output}} \) is the output node of \( Q \).

In these two data structures, all nodes are decreasingly sorted by their LeftPos values.

The algorithm is designed for queries containing \( l \)-edges, \( l \)-edges, \( * \), and branches. During the process, another algorithm \( \text{subsumption-check}(v, q) \) may be invoked to check whether any \( q \in q \) can be inserted into \( S(v) \), where \( q \) is a subset of query nodes such that \( L(q) \) contains \( v \). Let \( v_1, v_2 \) be two children of a node \( v \). Let \( S_1 = S(v_1) \) and \( S_2 = S(v_2) \). \( \text{merge}(S_1, S_2) \) puts \( S_1 \) and \( S_2 \) together with any duplicate being removed. Since both \( S_1 \) and \( S_2 \) are sorted by RightPos values, \( \text{merge}(S_1, S_2) \) works in a way like the sort-merge join and needs only \( O(\max(|S_1|, |S_2|)) \) time. We define \( \text{merge}(S_1, ..., S_{k-1}, S_k) \) to be \( \text{merge}(\text{merge}(S_1, ..., S_{k-1}), S_k) \).

The output of \( \text{tree-matching}( ) \) is \( D_{\text{root}} \) and \( D_{\text{output}} \). Based on them, we can find all the answers by generating a subtree in a way similar to the construction of \( T' \).

Algorithm \( \text{tree-matching}( ) \) does almost the same work as Algorithm \( \text{matching-tree-construction}( ) \). The main difference is lines 14 - 18 and lines 24 - 28. In lines 14 - 18, we set \( \chi \) values for some \( q \)'s. Each of them appears in a \( S(v) \), where \( v' \) is a child node of \( v \), satisfying the conditions i) - iii) given above. In lines 24 - 28, we use the merging operation to construct \( S(v) \). In Function \( \text{subsumption-check}( ) \), we check whether any \( q \) in \( q \) can be inserted into \( S \) by examining the ancestor-descendant/parent-child relationships (see line 4). For each \( q \) that can be inserted into \( Q \), we will further check whether it is the root of \( Q \).
or the output node of $Q$, and insert it into $D_{root}$ or $D_{output}$, respectively (see lines 6 - 8). In the Appendix, we prove the correctness of tree-matching().

#### Algorithm tree-matching($L(Q)$)

**Input:** all data streams $L(q)$.
**Output:** a matching subtree $T'$ of $T$, $D_{root}$ and $D_{output}$.

**Begin**
1. **repeat** until each $L(q)$ in $L(Q)$ becomes empty
2. **identify** $q$ such that the first node $v$ of $L(q)$ is of the minimal RightPos value; remove $v$ from $L(q)$; generate node $v$;
3. if $v$ is the first node created **then**
   4. $S(v) \leftarrow$ subsumption-check($v, q$);
5. **else**
   6. let $v'$ be the quadruple chosen just before $v$, for which a node is constructed;
    7. if $v'$ is not a child (descendant) of $v$ **then**
       8. left-sibling($v) \leftarrow v'$;
       9. $S(v') \leftarrow$ subsumption-check($v, q$);
      **else**
      10. $\{v'' \leftarrow v'; w \leftarrow v;\}$
      11. while $v''$ is a child (descendant) of $v$ **do**
         12. parent($v'') \leftarrow v; (* generate a parent link. Also, indicate whether $v''$ is a /-child or a I/-child.*)
         13. for each $q$ in $QS(v)$ **do**
            14. if $(q$ is a /-child) or
               15. $(q$ is a /-child and
                  16. $v''$ is a I/-child and
                  17. $\exists q''$ such that $v''$ is subsumed by $v''$)
                    then $\exists q''$ such that $v''$ is subsumed by $v''$;
                    $w \leftarrow v''$;
                    $v'' \leftarrow$ left-sibling($v''$);
                    remove left-sibling($w$);
                    20. **end**
                    21. **end**
                    22. left-sibling($v') \leftarrow v''$;
                    **end**
                    24. $S' \leftarrow$ subsumption-check($v, q$);
                    25. let $v_1, ..., v_n$ be the child nodes of $v$;
                    26. $S' \leftarrow$ merge($S(v_1), ..., S(v_n)$);
                    27. remove $S(v_1), ..., S(v_n)$;
                    28. $S(v) \leftarrow$ merge($S, S'$);
                    29. **end**
     **end**
**End**

**Function subsumption-check($v, q$)**

($v$ satisfies the node name test at each $q$ in $Q$.)

**Begin**
1. $S' \leftarrow \emptyset$
2. for each $q$ in $Q$ **do**
3. 3. let $q_1, ..., q_n$ be the child nodes of $q$.
4. 4. if for each I/-child $q_i$ $\gamma(q_i) = v$ and for each I/-child $q_i$
         $\gamma(q_i)$ is subsumed by $v$ **then**
5. 5. $S \leftarrow S \cup \{q\}$;
6. 6. if $q$ is the root of $Q$ **then**
7. 7. $D_{root} \leftarrow D_{root} \cup \{v\}$;
8. 8. if $q$ is the output node then $D_{output} \leftarrow D_{output} \cup \{v\}$;
9. **end**
**End**

Fig. 8. Algorithm tree-matching

The algorithm handles wildcards in the same way as any non-wildcard nodes. But a wildcard matches any tag name. Therefore, $L(*)$ should contain all the nodes in $T$. However, as with twigStack [5], we establish an XBT-tree over $B(q)$’s and take an element from it as it is needed. Recall that the input of our algorithm is in fact $B(q)$’s which are transformed to $L(q)$’s by using a global stack (see Fig. 5).

**Example 2** Applying Algorithm tree-matching to the data streams shown in Fig. 4, we will find that the document tree shown in Fig. 3 contains the query tree shown in Fig. 4. We trace the computation process as shown in Fig. 9.

In the first three steps, we will generate part of the matching subtree as shown in Fig. 9(a). Associated with $v_8$ is a query node stream: $QS(v_8) = \{q_3\}$. Although $q_2$ also matches $v_8$, it cannot survive the subsumption check (see line 4 in subsumption-check( )). So it does not appear in $QS(v_8)$. In addition, we have $QS(v_5) = QS(v_9) = \{q_3, q_4\}$. It is because both $q_3$ and $q_4$ are leaf nodes and can always satisfy the subsumption checking. In a next step, we will meet the parent $v_4$ (appearing in $L(q_2, q_5)$) of $v_5$ and $v_6$. So we are able to get $\gamma(q_2) = v_4$ and $\gamma(q_4) = v_4$ (see Fig. 9(b)). In terms of these two values, we know that $q_2$ should be inserted into $QS(v_4)$. $q_5$ is a leaf node and also inserted into $QS(v_4)$. In addition, $QS(v_5)$ and $QS(v_6)$ should also be merged into it. In the fifth step, we meet $v_3$. $QS(v_3) = \{q_3, q_4\}$ (see Fig. 9(c)). In the sixth step, we meet $v_2$ (in $L(q_2, q_5)$). It is the parent of $v_3$ and $v_4$. According to $QS(v_3) = \{q_3, q_4\}$ and $QS(v_4) = \{q_2, q_5\}$, as well as the fact that both $q_3$ and $q_4$ are I/-child nodes and $label(q_3) = label(q_4) = B$, we will set $\gamma(q_3) = \gamma(q_4) = \gamma(q_5) = v_2$ (see Fig. 9(d)). Thus, we have $QS(v_2) = \{q_2, q_3\}$. Finally, in step 7, according to $QS(v_2) = \{q_2, q_3\}$ and $QS(v_8) = \{q_5\}$, we will set $\gamma(q_2) = v_1$ and $\gamma(q_5) = v_1$ (see Fig. 9(e)), leading to the insertion of $q_1$ into $QS(v_1)$.

In Example 2, we see that if we just want to record only those parts of $T$, which contain the whole $Q$ or the subtree rooted at the output node, a $QS(v)$ can be removed once $v$’s parent is encountered. However, if we maintain them, we are able to tell all the possible containment, i.e., which parts of $T$ contain which parts of $Q$.

In the following, we prove the correctness of this algorithm.

**Lemma 1** Let $v_1$, $v_2$, and $v_3$ be three nodes in a tree with $v_3.RightPos > v_2.RightPos > v_1.RightPos$. If $v_1$ is a descendant of $v_3$. Then, $v_2$ must also be a descendant of $v_3$.

**Proof.** We consider two cases: i) $v_2$ is to the right of $v_1$, and ii) $v_2$ is an ancestor of $v_1$. In case (i), we have $v_1.LeftPos < v_2.LeftPos$. So we have $v_3.LeftPos < v_1.LeftPos < v_2.LeftPos$. This shows that $v_3$ is a descendant of $v_3$. In case (ii), $v_1$, $v_2$, and $v_3$ are on the same path. Since $v_2.LeftPos > v_3.LeftPos$, $v_2$ must be a
Fig. 10. Illustration for Lemma 1

Proposition 1 Let $Q$ be a twig pattern containing only $/$-edges, $\backslash$-edges and branches. Let $v$ be a node in the matching subtree $T'$ with respect to $Q$ created by Algorithm tree-matching(). Let $q$ be a node in $Q$. Then, $q$ appears in $QS(v)$ if and only if $T'[v]$ contains $Q[q]$.

Proof. If-part. A query node $q$ is inserted into $QS(v)$ by executing Function subsumption-check(), which shows that for any $q$ inserted into $QS(v)$ we must have $T'[v]$ containing $Q[q]$ for the following reasons:

1. $\text{label}(v) = \text{label}(q)$.
2. For each $\backslash$-child $q'$ of $q$ there exists a child $v'$ of $v$ such that $T'[v']$ contains $Q[q']$. (See line 15 in tree-matching().)
3. For each $\backslash$-child $q''$ of $q$ there exists a $\backslash$-child $v''$ of $v$ such that $T'[v'']$ contains $Q[q'']$ and $\text{label}(v'') = \text{label}(q'')$. (See lines 16--17 in tree-matching().)

In addition, a query node $q$ in $QS(v)$ may come from a $QS$ of some child node of $v$. Obviously, we have $T'[v]$ containing $Q[q]$.

Only-if-part. The proof of this part is tedious. In the following, we give only a proof for the simple case that $Q$ contains no $\backslash$-edges, which is done by induction of the height $h$ of the nodes in $T$.

Basis. When $h = 0$, for the leaf nodes of $T$, the proposition trivially holds.

Induction step. Assume that the proposition holds for all the nodes at height $h \leq k$. Consider the nodes $v$ at height $h = k + 1$. Assume that there exists a $q$ in $Q$ such that $T'[v]$ contains $Q[q]$ but $q$ does not appear in $QS(v)$. Then, there must be a child node $q_i$ of $q$ such that (i) $\chi(q_i) = \phi$, or (ii) $\chi(q_i)$ is not subsumed by $v$ when $q$ is checked against $v$. Obviously, case (i) is not possible since $T'[v]$ contains $Q[q]$ and $q_i$ must be contained in a subtree rooted at a node $v'$ which is a child (descendant) of $v$. So $\chi(q_i)$ will be changed to a value not equal to $\phi$ in terms of the induction hypothesis. Now we show that case (ii) is not possible, either. First, we note that during the whole process, $\chi(q_i)$ may be changed several times since it may appear in more than one $QS$'s. Assume that there exist a sequence of nodes $v_1, \ldots, v_k$ for some $k \geq 1$ with $v_1.\text{RightPos} < v_2.\text{RightPos} < \ldots < v_k.\text{RightPos}$ such that $q_i$ appears in $QS(v_1), \ldots, QS(v_k)$. In terms of the induction hypothesis, $v' = v_j$ for some $j \in [1, \ldots, k]$. Let $l$ be the largest integer $\leq k$ such that $v_l.\text{LeftPos} > v.\text{LeftPos}$. Then, for each $v_j (j \leq p \leq l)$, we have $v_j.\text{RightPos} < v.\text{RightPos}$.

In terms of Lemma 1, each $v_p (j \leq p \leq l)$ is subsumed by $v$. When we check $q$ against $v$, the actual value of $\chi(q_i)$ is the node name for some $v_p$'s parent, which is also subsumed by $v$ (in terms of Lemma 1), contradicting (ii). The above explanation shows that case (ii) is impossible. This completes the proof of the proposition.

The time complexity of the algorithm can be divided into three parts:

1. The first part is the time spent on accessing $L(Q)$. Since each element in a $L(Q)$ is visited only once, this part of cost is bounded by $O(|D| \cdot |Q|)$.
2. The second part is the time used for constructing $QS(v_j)$'s.
   For each node $v_j$ in the matching subtree, we need $O(\sum c_{i_j})$ time to do the task, where $c_{i_j}$ is the outdegree of $q_{i_j}$, which matches $v_j$. (See line 2 and 3 in Function subsumption-check() for explanation.) So this part of cost is bounded by $O(\sum c_{i_j}) \leq O(|D| \cdot \sum c_{i_j}) = O(|D| \cdot |Q|)$.
3. The third part is the time for establishing $\chi$ values, which is the same as the second part since for each $q$ in $QS(v)$ its $\chi$ value is assigned only once.

Therefore, the total time is $O(|D| \cdot |Q|)$.

The space overhead of the algorithm is easy to analyze. Besides the data streams, each node in the matching subtree needs a parent link and a right-sibling link to facilitate the subtree reconstruction, and an $QS$ to calculate $\chi$ values. So the extra space requirement is bounded by $O(|D| \cdot |Q| + |D| + |Q|) = O(|D| \cdot |Q|)$.

VI. ABOUT STRICT UNORDERED TREE MATCHING

Definition 2 A strict embedding of a tree pattern $Q$ into an XML document $T$ is a mapping $f: Q \rightarrow T$, from the nodes of $Q$ to the nodes of $T$, which satisfies the following conditions:

(i) same as (i) in Definition 1.
(ii) same as (ii) in Definition 1.
(iii) For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if $v_1$ and $v_2$ are not related by an ancestor-descendant relationship, then $f(v_1)$ and $f(v_2)$ in $T$ are not related by an ancestor-descendant relationship.

This is a much more difficult problem than the tree matching discussed in the previous section.

To facilitate the algorithm description, we will use some concepts from the hypergraph theory [1].

Definition 3 Let $U = \{u_1, \ldots, u_n\}$ be a finite set of nodes. A hypergraph on $U$ is a family $H = \{E_1, \ldots, E_l\}$ of subsets of $U$ such that

\[(1) \quad E_i \neq \emptyset \quad (i = 1, 2, \ldots, l)\]

\[(2) \quad \bigcup_{i=1}^{l} E_i = U.\]
A simple hypergraph (or Sperner family) is a hypergraph \( H = \{ E_1, \ldots, E_l \} \) such that

\[ E_i \subset E_j \Rightarrow i = j. \]

As for a graph \( H \), the order of \( H \), denoted by \( n(H) \), is the number of nodes. The number of edges will be denoted by \( m(H) \) and the rank of \( H \) is defined to be \( r(H) = \max_j |E_j| \). It can be proved that \( m(H) \leq \left\lfloor \frac{n}{n/2} \right\rfloor \) [1].

Let \( A \subset U \) be a subset. We call the family \( H_A = \{ E_i \cap A \mid 1 \leq i \leq l, E_i \cap A \neq \emptyset \} \) the sub-hypergraph induced by \( A \).

**Definition 4** Let \( H = \{ E_1, \ldots, E_l \} \) be a hypergraph on \( U \) and \( H' = \{ F_1, \ldots, F_r \} \) be another hypergraph on \( V \). The product of \( H \) and \( H' \), denoted as \( H \times H' \), is a hypergraph, whose nodes are the elements of the Cartesian product \( U \times V \), and whose edges are the sets \( E_i \times F_j \) with \( 1 \leq i \leq l \) and \( 1 \leq j \leq r \). Obviously, \( n(H \times H') = n(H)n(H') \) and \( m(H \times H') = m(H)m(H') \). However, if \( U = V \), we have \( n(H \times H') = n(H) \) and \( m(H \times H') \leq \left\lfloor \frac{n}{n/2} \right\rfloor \). □

As with the ordered tree embedding, we will maintain two matrices \( Q(P, T) \) and \( S(P, T) \) to control the computation.

1. In \( Q(P, T) \), an entry \( q_{ij} \) is 1 if the subtree rooted at \( j \) in \( T \) includes the subtree rooted at \( i \) in \( P \). Otherwise, it is 0.
2. In \( S(P, T) \), each entry \( s_{ij} \) is defined as follows. Let \( i_1, i_2, \ldots, i_k \) be the child nodes of \( i \). \( s_{ij} \) is a hypergraph \( H_i = \{ E_1, \ldots, E_l \} \) over \( i_1, i_2, \ldots, i_k \) such that \( T[j] \), the subtree rooted at \( j \), includes each \( E_{i_q} \) \( (g = 1, \ldots, l) \), that is, for each \( E_{i_q} \), the subtree rooted at \( j \) includes all the subtrees rooted at the nodes in \( E_{i_q} \). But \( T[j] \) cannot include \( E_i \cup E_j \) for any \( i, j \in \{ 1, \ldots, l \} \).

**Algorithm** unordered-embedding(\( T, P \))

Input: tree \( T \) (with nodes 1, ..., \( n \)) and tree \( P \) (with nodes 1, ..., \( m \))

Output: \( Q(P, T) \), which shows the tree embedding.

```
begin
1. for \( v := 1, \ldots, n \) do
2. {for \( u := 1, \ldots, m \) do
3. {if \( q_{uv} = 0 \) then
4. {let \( v_1, \ldots, v_k \) be the child nodes of \( v \);
5. \( H := s_{uv_1} \times s_{uv_2} \times \cdots \times s_{uv_k} \);
6. let \( u_1, \ldots, u_k \) be the child nodes of \( u \);
7. if \( \{ u_1, \ldots, u_k \} \in H \) then set \( q_{uv} \) to 1;
8. else \( s'_{uv} := H_A \);
9. }
10. }
11. let \( u_1, \ldots, u_l \) be nodes such that \( P[u_1], \ldots, P[u_l] \) are included in \( T[v] \);
12. for each ancestor \( v' \) of \( v \), \( q_{uv'} := 1 \) for \( i = 1, \ldots, l \);
13. construct hypergraph \( H = \{ \{ u_1 \}, \ldots, \{ u_l \} \} \);
14. for \( u := 1, \ldots, m \) do
15. {let \( A \) be the set of \( u \)'s child nodes;
16. \( s_{uv} := H_A \);
17. }
20. }
end
```

**V. EXPERIMENTS**

In this section, we report the test results. We conducted our experiments on a DELL desktop PC equipped with Pentium(R) 4 CPU 2.80GHz, 0.99GB RAM and 20GB hard disk. The code was compiled using Microsoft Visual C++ compiler version 6.0, running standalone.

- **Tested methods**
  - **TwigStack (TS for short)** [5],
  - **Twig2Stack (T2S for short)** [12],
  - **Twig-List (the method discussed in [15]), TL for short.**
  - **One-Phase Holistic (the method discussed in [42]; OPH for short),**
  - **Tree-matching (the method discussed in Section 3; TM for short),**
  and compare their execution times, as well as the runtime space usage. The theoretical computational complexities of these methods are summarized in Table 1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Query time</th>
<th>Runtime space usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwigStack</td>
<td>O(</td>
<td>D</td>
</tr>
<tr>
<td>Twig2Stack</td>
<td>O(</td>
<td>D</td>
</tr>
<tr>
<td>OPH</td>
<td>O(</td>
<td>D</td>
</tr>
<tr>
<td>TL</td>
<td>O(</td>
<td>D</td>
</tr>
<tr>
<td>TM</td>
<td>O(</td>
<td>D</td>
</tr>
</tbody>
</table>

**Table 1. Time and space complexities**

<table>
<thead>
<tr>
<th>Data set</th>
<th>TreeBank</th>
<th>DBLP</th>
<th>XMark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size</td>
<td>82 (MB)</td>
<td>127 (MB)</td>
<td>113 (MB)</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>2437k</td>
<td>3332k</td>
<td>1666k</td>
</tr>
<tr>
<td>Max/avg. depth</td>
<td>36/7.9</td>
<td>62/9.1</td>
<td>125/5</td>
</tr>
</tbody>
</table>

- **Data**
  The data sets used for the tests are TreeBank data set [30], DBLP data set [30] and a synthetic XMARK data set [35]. The TreeBank data set is a real data set with a narrow and deeply recursive structure that includes multiple recursive elements. The DBLP data set is another real data set with high similarity in structure. It is in fact a wide and shallow document. The XMark (with factor = 1) is a well-known benchmark data set. The important parameters of these data sets are summarized in Table 2.
- Queries

As we know, XPath allows for the formulation of straight-line queries as well as, in terms of XPath predicates, twigs that actually contain branches. XPath further allows the specification of value-based predicates. To study the performance impact of such characteristics, we have tested 10 queries against DBLP database, which are divided into two groups. In the first group all the 5 queries are with a constant while in the second group (another 5 queries) no parameter is specified. Over XMARK database, we have also tested 10 queries, divided into 2 groups with each containing 5 queries. In the first group, each query contains a constant. In the second group, for each query no constant is specified. All the queries are shown in Table 3 - Table 6.

Table 3: Group I - DBLP queries

<table>
<thead>
<tr>
<th>Query</th>
<th>XPath Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>//proceedings[author/* and title/*]year = '2004'</td>
</tr>
<tr>
<td>Q2</td>
<td>//proceedings[author and title*/year = '2004'</td>
</tr>
<tr>
<td>Q3</td>
<td>//proceedings[author and title *]year = '2004'</td>
</tr>
<tr>
<td>Q4</td>
<td>//articles[author and title *]pages and /year/text() = '2004'</td>
</tr>
<tr>
<td>Q5</td>
<td>//articles[author and title *]pages and /year/text() = '2004'</td>
</tr>
</tbody>
</table>

Table 4: Group II - DBLP queries

<table>
<thead>
<tr>
<th>Query</th>
<th>XPath Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q6</td>
<td>//proceedings[author* and /year]</td>
</tr>
<tr>
<td>Q7</td>
<td>//proceedings[author* and title *]year</td>
</tr>
<tr>
<td>Q8</td>
<td>//proceedings[author* and title *]year</td>
</tr>
<tr>
<td>Q9</td>
<td>//articles[author* and title *]pages and /year and /title/text() = '2004'</td>
</tr>
<tr>
<td>Q10</td>
<td>//articles[author* and title *]pages and /year and /title/text() = '2004'</td>
</tr>
</tbody>
</table>

Table 5: Group III - XMARK queries

<table>
<thead>
<tr>
<th>Query</th>
<th>XPath Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q11</td>
<td>/open_auction[.//seller/person/* and /date ]</td>
</tr>
<tr>
<td>Q12</td>
<td>/open_auction[.//seller/person.and /bidder]/date [text()] = '10/23/2006'</td>
</tr>
</tbody>
</table>
| Q13   | /open_auction[.//seller/person and /bidder/increase
date [text()] = '10/23/2006' |
| Q14   | /open_auction[.//seller/person and /bidder/increase
date [text()] = '10/23/2006' |
| Q15   | /open_auction[.//seller/person and /bidder/increase
date [text()] = '10/23/2006' |

Table 6: Group IV - XMark queries

<table>
<thead>
<tr>
<th>Query</th>
<th>XPath Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q16</td>
<td>/open_auction[.//seller/person* and /date ]</td>
</tr>
<tr>
<td>Q17</td>
<td>/open_auction[.//seller/person* and /bidder and /date ]</td>
</tr>
<tr>
<td>Q18</td>
<td>/open_auction[.//seller/person* and /bidder/increase]</td>
</tr>
<tr>
<td>Q19</td>
<td>/open_auction[.//seller/person* and /bidder/increase and /initial and /date ]</td>
</tr>
<tr>
<td>Q20</td>
<td>/open_auction[.//seller/person* and /bidder/increase and /initial and /description and /date ]</td>
</tr>
</tbody>
</table>

- Test results

Now we demonstrate the execution times of all the four strategies when they are applied to the above queries. In Fig. 11(a), we show the test results of the first group. From these we can see that our algorithm outperforms all the other strategies. It is because this algorithm works only in one scan of the data streams and neither the path join nor the result enumeration is involved. More importantly, for each element from an XB-tree, our algorithm only checks Q5s for the child nodes of current query node. But Twig^2 Stack needs to check all the stacks associated with all the query nodes. Both OPH and TL have the same problems Twig^2 Stack, but work a little bit better than Twig^2 Stack. OPH does no result enumeration is involved while TL does less checkings. TwigStack has the worst performance.

Fig. 11. Results of Group I and II

Fig. 11(b) shows the test results of the second group. The execution time of all the strategies are much worse than Group I since the queries are all of quite low selectivity and thus almost all the data set has to be downloaded into main memory. In this case, I/O dominates the cost. Again, our algorithm has the best performance. Especially, when the size of queries becomes larger, this algorithm is 3 - 4 times better than Twig^2 Stack, TL and OPH. First, the time for constructing a matching subtree is much less than that for constructing the hierarchical stacks. Secondly, the space used by our algorithm is much smaller than any of the three methods. TwigStack shows an exponential-time behavior since for each path in a query a great many matching paths will be produced and the cost of join operations increases exponentially.

Fig. 12. Results of Group III and IV

In Fig. 12(a) and (b), the test results over the XMARK database are demonstrated. From these, we can see that our algorithm still has the best performance for this data set.

In Fig. 13, we compare the runtime memory usage of all the four tested approaches for the second group of queries.

Fig. 13. Runtime memory usage of Group II

By the memory usage, we mean the intermediate data structures, not including data stream (concretely, path stacks
for TwigStack; hierarchical stacks for Twig^2 Stack, TL and OPH; and QSs for ours.)

VI. CONCLUSION

In this paper, a new algorithms is presented to evaluate twig pattern queries based on unordered tree matching. The main idea is a process for tree reconstruction from data streams, during which each node v that matches a query node will be inserted into a tree structure and associated with a query node stream QS(v) such that for each node q in QS(v) T[v] embeds Q[q]. Especially, by using an important property of the tree encoding, this process can be done very efficiently, which enables us to reduce the time complexity of the existing methods (such as Twig^2 Stack [12], Twig-List [15], and One-Phase Holistic [42]) by one order of magnitude. Our experiments demonstrate that the new algorithm is both effective and efficient for the evaluation of twig pattern queries.

REFERENCES