UNORDERED TREE MATCHING AND TREE PATTERN QUERIES IN XML DATABASES

Yangjun Chen
Dept. Applied Computer Science, University of Winnipeg, Manitoba, R3B 2E9, Canada

Keywords: Tree mapping, Tree pattern queries, XML databases, Query evaluation, Tree encoding.

Abstract: With the growing importance of XML in data exchange, much research has been done in providing flexible query facilities to extract data from structured XML documents. In this paper, we discuss an efficient algorithm for tree mapping problem in XML databases based on unordered tree matching. Given a target tree \( T \) and a pattern tree \( Q \), the algorithm can find all the embeddings of \( Q \) in \( T \) in \( O(|D| \cdot |Q|) \) time, where \( D \) is a largest data stream associated with a node of \( Q \). More importantly, the algorithm is index-oriented: with \( XB \)-trees constructed over data streams, disk access can be dramatically decreased.

1 INTRODUCTION

In this paper, we consider a kind of tree mappings used in XML databases, in which a set of XML documents is maintained. Abstractly, each document can be considered as a tree structure with each node standing for an element name from a finite alphabet \( \Sigma \); and an edge for the element-subelement relationship. Therefore, queries in XML query languages, such as XPath (Deutch et al., 1999), XQuery (Wang et al., 2003; Wang et al., 2005), XML-QL (Cooper et al., 2001), and Quilt (Chamberlin et al., 2000; Chamberlin et al., 2002), typically specify patterns of selection predicates on multiple elements that also have some specified tree structured relations. For instance, the XPath expression:

\[
\text{book}[\text{title} = 'Art of Programming']//\text{author}[\text{fn} = 'Donald' \text{ and } \text{ln} = 'Knuth']
\]

matches \textit{author} elements that (i) have a child subelement \( \text{fn} \) with content 'Donald', (ii) have a child subelement \( \text{ln} \) with content 'Knuth', and are descendants of \textit{book} elements that have a child \textit{title} subelement with content 'Art of Programming'.

This expression can be represented as a tree structure as shown in Fig. 1.

In this tree structure, the nodes \( v \) are labeled with element names or string values, denoted as \( \text{label}(v) \). In addition, there are two kinds of edges: child edges (/-edges) for parent-child relationships, and descendant edges (//edges) for ancestor-descendant relationships. A /-edge from node \( v \) to node \( u \) is denoted by \( v \rightarrow u \) 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 addition, a node in \( Q \) can be a wildcard '*' that matches any element in \( T \). Such a query is often called a twig pattern.

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 a twig pattern, 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 twig 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) Preserve node label: For each \( u \in Q \), \( \text{label}(u) = \text{label}(f(u)) \).

(ii) Preserve parent-child/ancestor-descen-
If there exists a mapping from $Q$ into $T$, we say, $Q$ can be imbedded into $T$, or say, $T$ contains $Q$.

Notice that an embedding could map several nodes with the same tag name in a query to the same node in a database. It also allows a tree mapped to a path. In fact, it is a kind of unordered tree matching, by which the order of siblings is not significant. This definition is quite different from the tree matching defined in (Hoffman and O’Donnell, 1982).

In the past decade, there is much research on how to find such a matching efficiently; but all the proposed methods can be categorized into two groups. By the first group (Abiteboul et al., 1999; Chung et al., 2002; Chen et al., 2006), a tree pattern is typically decomposed into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations. Then, an index structure is used to find all the matching pairs that are joined together to form the final result. By the second group (Bruno et al., 2002; Chen et al., 2005; Choi et al., 2003; Lu et al., 2005; Seo et al., 2003; Li et al., 2001), a twig pattern is decomposed into a set of paths. The final result is constructed by joining all the matching paths together. As an important improvement, TwigStack was proposed by Bruno et al. (2002), which compresses the intermediate results by the stack encoding, which represents in linear space a potentially exponential number of answers. However, TwigStack achieves optimality only for the queries that contain only // -edges. In the case that a query contains both / -edges and // -edges, some useless path matchings have to be performed. In the worst case, TwigStack needs $O(|D||Q|)$ time for doing the merge joins as shown by Chen et al. See page 287 in (Chen et al., 2006). Here, $D$ is a largest data stream associated with a node $q$ of $Q$ and each element in a data stream is a quadruple $(DocID, LeftPos, RightPos, LevelNum)$ representing an element $v$ (matching $q$) in a document, where DocID is the document identifier; LeftPos and RightPos are generated by counting word numbers from the beginning of the document until the start and end of $v$, respectively; and LevelNum is the nesting depth of $v$ in the document. This method is further improved by several researchers. In (Chen et al., 2005), iTwigJoin was discussed, which exploits different data partition strategies. In (Lu et al., 2005), TIFast accesses only leaf nodes by using extended Dewey IDs. By both methods, however, the path joins can not be avoided.

The method TwigStack proposed by Chen et al. (2006) works in a quite different way. It represents the twig results using the so-called hierarchical stack encoding to avoid any possible useless path matchings. In (Chen et al., 2006), it is claimed that TwigStack needs only $O(|D||Q| + |subTwigResults|)$ time for generating paths. But a careful analysis shows that the time complexity for this task is actually bounded by $O(|D||Q|^2 + |subTwigResults|)$. It is because each time a node is inserted into a stack associated with a node in $Q$, not only the position of this node in a tree within that stack has to be determined, but a link from this node to a node in some other stack has to be constructed, which requires to search all the other stacks in the worst case. The number of these stacks is $|Q|$. See Fig. 4 in (Chen et al., 2006) to know the working process. The following example helps for illustration.

![Figure 2: Illustration for hierarchical stacks.](image-url)

In Fig. 2(b), we show the hierarchical stacks associated with the two nodes $A$ and $B$ of $Q$ with respect to $T$ shown in Fig. 2(a). In (Chen et al., 2006), the nodes in a data stream associated with each node of $Q$ are sorted by their (DocID, RightPos) values. So $a_1$ is visited last. When it is inserted into HS[A] (hierarchical stack of $A$), all those stacks in HS[A], which are not a descendant of some other stack, will be checked to establish ancestor-descendant links. In addition, to generate links to some stacks in HS[B], similar checks will also be performed. This needs $O(|Q|)$ time in the worst case, yielding an $O(|D||Q|^2)$ time complexity. The method discussed in (Jiang et al., 2007) improves the stack structure used in TwigStack to avoid storing individual path matches and remove subTwigResults time. But its theoretical time complexity is still $O(|D||Q|^2)$.

In this paper, we present a new algorithm, tree-matching( ), for evaluating tree pattern queries with the following advantages:

- tree-matching( ) is able to handle twig patterns containing / -edges, // -edges, *, and branches.
- tree-matching( ) takes a set of data streams as inputs, over which XB-trees can be established to speed up disk access.
- tree-matching( ) runs in \(O(|D| |Q|)\) time and \(O(|D| |Q|)\) space.

The remainder of the paper is organized as follows. In Section 3, we restate the tree encoding (Zhang et al., 2001), which facilitates the recognition of different relationships among the nodes of a tree. In Section 3, we discuss our algorithm. Section 4 is devoted to the adaptation of our algorithm in an indexing environment. Finally, a short conclusion is given in Section 5.

2 TREE ENCODING

In (Zhang et al., 2001), 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 \(\alpha(v) = (d, l, r, ln)\), where \(d\) is the document identifier (DocID), \(l = \text{LeftPos}\), \(r = \text{RightPos}\), and \(ln = \text{LevelNum}\). By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined (Zhang et al., 2001):

(i) ancestor-descendant: a node \(v_1\) associated with \((d_1, l_1, r_1, ln_1)\) is an ancestor of another node \(v_2\) with \((d_2, l_2, r_2, ln_2)\) iff \(d_1 = d_2, l_1 < l_2,\) and \(r_1 > r_2\).

(ii) parent-child: a node \(v_1\) associated with \((d_1, l_1, r_1, ln_1)\) is the parent of another node \(v_2\) with \((d_2, l_2, r_2, ln_2)\) iff \(d_1 = d_2, l_1 < l_2, r_1 > r_2,\) and \(ln_2 = ln_1 + 1\).

(iii) from left to right: a node \(v_1\) associated with \((d_1, l_1, r_1, ln_1)\) is to the left of another node \(v_2\) with \((d_2, l_2, r_2, ln_2)\) iff \(d_1 = d_2, r_1 < l_2\).

\[
\begin{align*}
T: & \quad A \rightarrow v_1 \quad (1, 1, 11, 1) \\
& \quad \quad B \rightarrow v_2 \quad (1, 3, 3, 3) \\
& \quad \quad \quad v_2 \vee C \quad (1, 3, 5, 4) \\
& \quad \quad \quad \quad B \rightarrow v_3 \quad (1, 4, 8, 3) \\
& \quad \quad \quad \quad \quad v_3 \vee C \quad (1, 6, 6, 4) \\
& \quad \quad \quad \quad \quad \quad D \rightarrow v_4 \quad (1, 7, 7, 4)
\end{align*}
\]

Figure 3: Illustration for tree encoding.

In Fig. 3, \(v_2\) is an ancestor of \(v_6\) and we have \(v_2.\text{LeftPos} = 2 < v_6.\text{LeftPos} = 6\) and \(v_2.\text{RightPos} = 9 > v_6.\text{RightPos} = 6\). In the same way, we can verify all the other relationships of the nodes in the tree. In addition, for each leaf node \(v\), we set \(v.\text{LeftPos} = v.\text{RightPos}\) for simplicity, which still work without downgrading the ability of this mechanism.

In the rest of the paper, if for two quadruples \(\alpha_1 = (d_1, l_1, r_1, ln_1)\) and \(\alpha_2 = (d_2, l_2, r_2, ln_2)\), we have \(d_1 = d_2, l_1 < l_2,\) and \(r_1 > r_2,\) we say that \(\alpha_2\) is subsumed by \(\alpha_1\). For convenience, a quadruple is considered to be subsumed by itself. If no confusion is caused, we will use \(v\) and \(\alpha(v)\) interchangeably.

We can also assign LeftPos and RightPos values to the query nodes in \(Q\) for the same purpose as above. Finally we use \(T[v]\) to represent a subtree rooted at \(v\) in \(T\).

3 MAIN ALGORITHM

In this section, we discuss our algorithm according to Definition 1. The input of the algorithm is a set of data streams associated with the query nodes \(q\) in \(Q\), which contains the positional representations (quadruples) of the document 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, RightPos) 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.

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. Without loss of generality, assume that the query nodes in \(q\) are sorted by their RightPos values.

\[
Q: \quad \text{Query nodes with the same tag will be associated with the same data stream:} \\
L(q) = L(q_1) = L(q_2) = \{v_1, v_2, v_3\}
\]

Figure 4: Illustration for \(L(q)\)’s.

We will also use \(L(Q) = \{L(q_1), \ldots, L(q_l)\}\) 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. During the process, for each document tree node \(v\), a data structure is produced and maintained to facilitate computation:

\(QS(v)\) - it contains all those query tree node \(q\) such that \(Q[q]\) (the subtree rooted at \(q\)) can be imbedded into \(T[v]\).
In addition, each \( q \) is associated with a variable, denoted \( \chi(q) \). During the tree matching process, \( \chi(q) \) will be dynamically assigned a series of values \( a_{i-1}, a_i, \ldots, a_m \) for some \( m \) in sequence, where \( a_0 = \phi \) and \( a_i/s \) (i = 1, ..., 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, ..., m) when the following conditions are satisfied.

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

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

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

\( q \) is a /-child, and \( u \) is a /-child with \( label(u) = label(q) \).

Then, each time before we insert \( q \) into \( QS(v) \), 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, ..., k), \( \chi(q_i) \) is equal to \( v \) and \( label(v) = label(q_i) \), insert \( q \) into \( QS(v) \).

Since we search both \( T \) and \( Q \) bottom-up, the above checking guarantees that for any \( q \in QS(v) \), \( T[v] \) contains \( Q[q] \).

Below we show our algorithm \textit{tree-matching}(L(Q)) for queries containing /-edges, //edges, *, and //edges. During the execution, another algorithm \textit{subsumption-check}(v, q) may be invoked to check whether any \( q \in QS(v) \) can be inserted into \( QS(v) \).

In the whole process, the quadruples will be removed one by one from the data streams and for each of them a node will be created and inserted into a temporary tree structure, called a matching subtree.

\textbf{Algorithm tree-matching}(L(Q))

\textbf{Input:} all data streams \( L(Q) \)

\textbf{Output:} a matching subtree \( T' \) of \( T \), \( D_{root} \) and \( D_{output} \).

\begin{algorithm}
\caption{Tree-Matching Algorithm}
\begin{algorithmic}
\State \textbf{begin}
\Repeat \text{until each } L(q) \text{ in } L(Q) \text{ becomes empty }
\State \text{identify } v \text{ such that the first node } v \text{ of } L(q) \text{ is of the minimal RightPos value; remove } v \text{ from } L(q); \text{ generate node } v;
\If {v is the first node created}
\State \{ QS(v) ← subsumption-check(v, q); \}
\Else 
\State \{ let } v' \text{ be the quadruple chosen just before } v, \text{ for which a node is constructed; }
\If {v' is not a child (descendant) of } v \text{ then }
\State \{ left-sibling(v) ← v'; (*generate a left-sibling link from } v \text{ to } v'.\}
\State QS(v) ← subsumption-check(v, q);
\Else 
\State \{ v'' ← v'; w ← v'; (*v'' and w are two temporary units.*\}
\While {v'' is a child (descendant) of } v 
\State \{ parent(v'') ← v; (*generate a parent link. Also, indicate whether } v'' \text{ is a /-child or a //child.*) \}
\EndWhile 
\EndIf
\EndIf
\EndRepeat 
\State \text{for each } q \text{ in } QS(v') \text{ do } \}
\If {(q is a /-child) or (q is a //child and } v'' \text{ is a /-child and }
\State \{ label(q) = label(v'')\} 
\Then \chi(q) ← v' ; 
\State \{ w ← v''; v'' ← left-sibling(v''); \}
\State remove left-sibling(w); \}
\Else 
\State { left-sibling(v) ← v''; \}
\EndIf
\EndFor 
\State \{ q ← subsumption-check(v, q); \}
\State \{ let } v_1, \ldots, v_m \text{ be the child nodes of } v; \}
\State \{ q' ← merge(QS(v_1), \ldots, QS(v_m)); \}
\State remove QS(v_1), \ldots, QS(v_m); \}
\State \{ QS(v) ← merge(q, q'); \}
\EndFunction
\end{algorithm}
\end{algorithm}

The outputs of the above algorithm are mainly two data structures:

\begin{itemize}
\item \( D_{root} \) - a subset of document nodes \( v \) such that \( Q \) can be embedded in \( T[v] \).
\item \( D_{output} \) - a subset of document nodes \( v \) such that \( Q[q_{output}] \) can be embedded in \( T[v] \), where \( q_{output} \) is the output node of \( Q \).
\end{itemize}

In these two data structures, all nodes are increasingly sorted by their RightPos values. Based on them, we can find all the answers.

In addition, special attention should be paid to \textit{merge}(QS_1, QS_2), which puts QS_1 and QS_2 together with any duplicate being removed. Since both \( QS_1 \) and \( QS_2 \) are sorted by RightPos values, \textit{merge}(QS_1, QS_2) works in a way like the sort-merge join and takes only \( O(\max\{|QS_1|, |QS_2|\}) \) time. We define \textit{merge}(QS_1, ..., QS_k, 1), \( QS_k \) to be \textit{merge}(merge(QS_1, ..., QS_{k-1}), QS_k).

In lines 14 - 18, we set \( \chi \) values for some \( q \)'s. Each of them appears in a \( QS(v') \), 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 \( QS(v) \).

\textbf{Function subsumption-check}(v, q) (*\( v \) satisfies the node name test at each } q \text{ in } q.*\)

\begin{algorithm}
\caption{Subsumption-Check Function}
\begin{algorithmic}
\State \{ QS ← \;
\State \{ for each } q \text{ in } q \text{ do } \}
\State \{ let } q_1, \ldots, q_k \text{ be the child nodes of } q; \}
\State \{ if for each } /-child \text{ q}_i \text{, } \chi(q_i) = v \text{ and for each } //child \text{ q}_i \text{, } \chi(q_i) \text{ is subsumed by } v \text{ then }
\State \{ QS ← QS \cup \{ q \}; \}
\State \{ if } q \text{ is the root of } Q \text{ then }
\State \{ D_{root} ← D_{root} \cup \{ v \}; \}
\State \{ if } q \text{ is the output node then } D_{output} ← D_{output} \cup \{ v \}; \}
\State \{ return } QS; \}
\EndFunction
\end{algorithm}
\end{algorithm}

In Function \textit{subsumption-check}( ), we check whether any \( q \) in \( q \) can be inserted into \( QS \) by examining the
ancestor-descendant/parent-child relationships (see line 4). For each \( q \) that can be inserted into \( QS \), we will further check whether it is the root of \( Q \) or the output node of \( Q \), and insert it into \( D_{\text{roo}} \) or \( D_{\text{out}} \) respectively (see lines 6 - 8).

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 (Bruno, et al., 2002), we establish an XB-tree over the data stream and take an element from it as it is needed. We discuss this issue in Section 4.

**Example 1.** 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. 5.

 proclaimed over \( B(q) \), instead of \( L(q) \).

Algorithm stream-transformation(\( B(q) \)’s).

input: all data streams \( B(q) \)’s, each sorted by LeftPos.

output: new data streams \( L(q) \)’s, each sorted by RightPos.

begin
1. repeat until \( B(q) \) becomes empty
2. identify \( q \), such that the first element \( v \) of \( B(q) \) is of the minimal LeftPos value; remove \( v \) from \( B(q) \); 
3. while \( ST \) is not empty and \( ST.top \) is not \( v \)’s ancestor
do
4. \( x \leftarrow ST.pop() \); Let \( x = (q, v) \);
5. put \( u \) at the end of \( L(q) \); 
6. \( ST.push(q, v) \);
7. \( ST.push(q, v) \);
8. \} end

In the above algorithm, \( ST \) is used to keep all the nodes on a path until we meet a node \( v \) that is not a descendant of \( ST.top \). Then, we pop up all those nodes that are not \( v \’s \) ancestor; put them at the end of the corresponding \( L(q) \)’s (see lines 3 - 4); and push \( v \) into \( ST \) (see line 7). The output of the algorithm is a set of data streams \( L(q) \)’s with each being sorted by RightPos values. However, we remark that the popped nodes are in postorder. So we can directly handle the nodes in this order without explicitly generating \( L(q) \)’s. That is, in the main loop of Algorithm tree-matching( ), we handle the popped nodes one by one.

In the XB-tree established over an \( B(q) \), each entry in a page is a pair \( a = (\text{LeftPos}, \text{RightPos}) \) (referred to as a bounding segment) such that any entry appearing in the subtree pointed to by the pointer associated with \( a \) is subsumed by \( a \). In addition, all the entries in a page are sorted by their LeftPos values. As an example, consider a sorted quadruple sequence shown in Fig. 6(a), for which we may generate an XB-tree as shown in Fig. 6(b).

In each page \( P \) of an XB-tree, the bounding segments may partially overlap, but their LeftPos positions are in increasing order. Besides, it has two extra data fields: \( P.parent \) and \( P.parentIndex \). \( P.parent \) is a pointer to the parent of \( P \), and \( P.parentIndex \) is a number \( i \) to indicate that the \( i \)th pointer in \( P.parent \) points to \( P \). For instance, in the
XB-tree shown in Fig. 6(b), \( P_i, parentIndex = 2 \) since the second pointer in \( P_1 \) (the parent of \( P_3 \)) points to \( P_2 \).

We notice that in a \( Q \) we may have more than one query nodes \( q_1, ..., q_k \) with the same label. So they will share the same data stream and the same XB-tree. For each \( q_j \) (\( j = 1, ..., k \)), we maintain a pair (\( P, i \), denoted \( q \)), to indicate that the \( i \)th entry in the page \( P \) is currently accessed for \( q \). Thus, each \( j \) (\( j = 1, ..., k \)) corresponds to a different searching of the same XB-tree as if we have a separate copy of that XB-tree over \( B(q) \).

In (Bruno et al., 2002) two operations are defined to navigate an XB-tree, which change the value of \( \beta_q \).

1. **advance\( (\beta_q) \)** (going up from a page to its parent):
   
   \[
   \beta_q = (P, i) \noindent \text{if } \beta_q = (P, i) \text{ does not point to the last entry of } P, \quad i \leftarrow i + 1.
   \]
   
   Otherwise, \( \beta_q \leftarrow (P, parent, P.parentIndex + 1) \).

2. **drilldown\( (\beta_q) \)** (going down from a page to one of its children):
   
   \[
   \beta_q = (P', 1), \quad \text{where } P' \text{ is the } i \text{th child page of } P.
   \]

Initially, for each \( q \), \( \beta_q \) points to (\( rootPage, 0 \)), the first entry in the root page. We finish a traversal of the XB-tree for \( q \) when \( \beta_q = (rootPage, last) \), where \( last \) points to the last entry in the root page, and we advance it (in this case, we set \( \beta_q \) to \( \phi \), showing that the XB-tree over \( B(q) \) is exhausted.) As with \( \text{TwigStackXB} \), the entries in \( B(q)'s \) will be taken from the corresponding XB-tree; and many entries can be possibly skipped. Again, the entries taken from XB-trees will be reordered as shown in Algorithm stream-transformation(). According to (Bruno et al., 2002), each time we determine a \( q \in Q \), for which an entry from \( B(q) \) is taken, the following three conditions are satisfied:

i) For \( q \), there exists an entry \( v \) in \( B(q) \) such that it has a descendant in each of the streams \( B(q) \) (where \( q \) is a child of \( q \).

ii) Each recursively satisfies (i).

iii) \( \text{LeftPos}(v) \) is minimum.

In the case of XB-trees, we modify the function \( \text{getNext}(q) \) given in (Bruno et al., 2002) to do the task and fit it for our strategy, in which the following functions are used:

- \( \text{isLeaf}(q) \) - returns \( true \) if \( q \) is a leaf node of \( Q \), otherwise, \( false \).
- \( \text{currL}(\beta_q) \) - returns the leftPos of the entry pointed to by \( \beta_q \).
- \( \text{currR}(\beta_q) \) - returns the rightPos of the entry pointed to by \( \beta_q \).
- \( \text{isPlainValue}(\beta_q) \) - returns \( true \) if \( \beta_q \) is pointing to a leaf node in the corresponding XB-tree.

**Function:getNext\( (q) \)** (*Initially, \( q \) is the root of \( Q \).*)

begin
1. if (isLeaf\( (q) \)) then return \( q \);
2. for each child \( q_i \) of \( q \) do
3. \{ \( r_i \leftarrow \text{getNext}(q_i) \); \}
4. if (there exists at least an \( r_i \) such that \( r_i \neq q_j \)) then return \( q \);
5. else return \( q \);
end

The goal of the above function is to figure out a query node to determine what entry from data streams will be checked in a next step, which has to satisfy the above conditions (i) - (iii). So the algorithm works in a recursive way (see line 3 and condition (ii).) Lines 6 - 9 are used to find a query node satisfying condition (i). Lines 4, 5, 9 and 10 show that condition (iii) must be met. Special attention should be paid to line 5. We may have more than one \( r_i \)’s with the same minimal \( currL() \). In this case, the one with the maximum RightPos is returned. It is because the access sequence of the document nodes will be reordered. This arrangement enables us to check query nodes (against a certain document node) in postorder.

Based on the above algorithm, \( \text{tree-matching}(v) \) is extended to \( \text{tree-matchingXB}(v) \) with \( \beta_q \)'s being used to navigate different XB-trees, which is controlled by a specific procedure called \( \text{XB-navigation}(v) \) (see below). In addition, for each created node \( v \) in \( T' \), both \( S \) and \( A \) are handled as global variables. For each \( q, R_q \) is also a global variable such that for each \( v \in R_q \), \( T'[v] \) embeds \( Q(q) \).

**Algorithm: tree-matchingXB\( (Q) \)**

begin
1. while (\( \neg \text{end}(Q) \)) do
2. \{ \( q \leftarrow \text{getNext}(\text{rootPage}; Q) \); \}
3. if (isPlainValue\( (\beta_q) \)) then
4. \{ \( \text{let } v \text{ be the node pointed to by } \beta_q \); \}
5. while \( ST \) is not empty and \( ST.top \) is not \( v \)'s ancestor do
6. \{ \( x \leftarrow ST.pop(); \text{ Let } x = (q', u); \quad (* \text{a node for } u \text{ will be created.}) \}
7. \{ call \text{embeddingCheck}(q', u); \}
8. \{ \text{push}(q, v); \text{ advance}(\beta_q); \}
9. \}
end
In the above algorithm, all the entries from data streams will be visited through XB-trees (see line 3 and 10.) But they will be reordered by using a global stack $ST$ so that they are handled actually in postorder (see lines 4 - 9; also see Algorithm stream-transformation() for comparison.) For checking the tree embedding, Algorithm embeddingCheck() is invoked (see line 7) while for navigating an XB-tree Algorithm XB-navigation() is called (see line 10.)

**Procedure** XB-navigation($q$)

Input: a query node $q$.
Output: $\beta_v$ is changed.

\begin{enumerate}
\item $q$ is the first node (in postorder) then drilldown($\beta_q$);
\item $q$ is the left of $q$ then 
\begin{enumerate}
\item if empty($R_q$) $\land$ currL($\beta_q$) $>$ currR($\beta_q$) then advance($\beta_q$) (*not part of a solution*)
\item else drilldown($\beta_q$) (*may have a child in some solution*)
\end{enumerate}
\item $q$ is the parent of $q'$ (*$*$)
\item if $\neg$ empty($R_q$) $\lor$ currL($\beta_q$) $>$ currR($\beta_q$) then drilldown($\beta_q$)
\item else advance($\beta_q$);
\end{enumerate}

\end{enumerate}

The above procedure shows a way different from TwigStackXB to control the navigation of XB-trees. On the one hand, it is because we check the tree embedding bottom-up. On the other hand, we use not only ancestor-descendant, but also left-to-right relationships to control the XB-tree traversal. First, we examine whether $q$ is the first node in postorder (see line 1.) If it is the case, we will drill down the corresponding XB-tree since along the branch we may find some entries which are part of a solution. In general, we will check the query node $q'$ which is the predecessor of $q$ in postorder. It can be to the left of $q$ or the right-most child of $q$. In the former case, we will compare currL($\beta_q$) and currR($\beta_q$). If empty($R_q$) and currL($\beta_q$) $>$ currR($\beta_q$), any entry in the subtree rooted at the entry pointed to by $\beta_q$ cannot be part of a solution, so $\beta_q$ will be advanced (see lines 4 - 5.) Otherwise, we will drill down the subtree to find some entries which might be part of a solution (see line 6.) A similar analysis applies to lines 7 - 10.

**Procedure** embeddingCheck($q, v$)

Input: a query nodes $q$; a document tree node $v$.
Output: a matching subtree $T'$ of $T$, $D_{\text{root}}$ and $D_{\text{output}}$.

\begin{enumerate}
\item generate node $v$;
\item (*same as lines 3 – 29 in tree-matching*)
\end{enumerate}

\section{5 CONCLUSIONS}

In this paper, a new algorithm 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 $QSV(v)$ such that for each node $q$ in $QSV(v)$ $T[v]$ 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 (Chen et al., 2006) and One-Phase Holistic (Jiang et al., 2007) by one order of magnitude. Our experiments demonstrate that the new algorithm is both effective and efficient for the evaluation of twig pattern queries.

\section{REFERENCES}

Abiteboul, S., Buneman, P. and Suciu, D., 1999. Data on the web: from relations to semistructured data and XML, Morgan Kaufmann Publisher, Los Altos, CA 94022, USA.


Chung, C., Min, J. and Shim, K., 2002. APEX: An adaptive path index for XML data, ACM SIGMOD.


McHugh, J. and Widom, J., 1999. Query optimization for XML, in Proc. of VLDB.


