A New Algorithm for Evaluating Ordered Tree Pattern Queries

Yangjun Chen
Dept. Applied computer Science
University of Winnipeg, Winnipeg, Manitoba, Canada, R3b 2E9

Abstract **-** *An XML tree pattern query, represented as a labeled tree, is essentially a complex selection predicate on both structure and content of an XML. Tree pattern matching has been identified as a core operation in querying XML data. We distinguish between two kinds of tree pattern matchings: ordered and unordered tree matching. By the unordered tree matching, only ancestor-descendant and parent-child relationships are considered. By the ordered tree matching, the order of siblings is also taken into account. While different fast algorithms for unordered tree matching are available, no efficient algorithm for ordered tree matching for XML data exists. In this paper, we discuss a new algorithm for processing ordered tree pattern queries, whose time complexity is polynomial. In addition, the algorithm can be adapted to an indexing environment with XB-trees being used. Experiments have been conducted, which shows that the new algorithm is promising.*

Keywords: XML data stream, tree pattern queries, ordered tree matching

1 Introduction

In XML [43, 44], data is represented as a tree; associated with each node of the tree is an element name from a finite alphabet Σ. The children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element.

Accordingly, in most of the XML query languages (e.g. *XPath* [43], *XQuery* [44], *XML-QL* [15], and *Quilt* [6, 7]), queries are typically expressed by tree patterns (for example, path expressions expressed in XPath, path expressions in the *for* and *let* clauses in XQuery.) In such tree patterns, nodes are labeled with symbols from $\Sigma \cup \{*\}$ (* 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(a), which 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:

$a[b[c$ and $.//f]]/b[c$ and $e//d]$.

Fig. 1(a), 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 $v \rightarrow u$ in the text, and represented by a single arc; u is called a */-child* of v. A //-edge is denoted by $v \Rightarrow u$ in the text, and represented by a double arc; u is called a *//-child* of v.

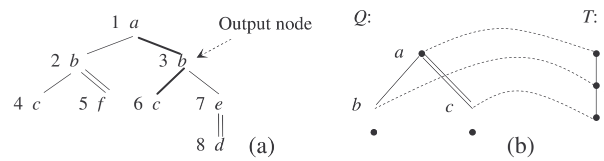

Fig. 1. A query tree and a tree matching a path

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 tree pattern, this path could consist of any sequence of /-edges and/or //-edges. We also use *label*(v) to represent the symbol ($\in \Sigma \cup \{*\}$) 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: $Q \rightarrow T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) Preserve node label: For each $u \in Q$, *label*(u) = *label*(f(u)) (or say, u matches f(u)).

(ii) Preserve *parent-child*/*ancestor-descendant* relationship: If $u \rightarrow v$ 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 imbedded into T, or say, T contains Q.

Almost all the existing strategies for evaluating twig join patterns are designed according to this definition [4, 8, 10, 11, 12, 14, 24, 26, 28, 29, 30, 31, 36, 37, 46].

This definition allows a path to match a tree as illustrated in Fig. 1(b).

It is because by Definition 1 the left-to-right relationships between siblings are not taken into account. We call such a problem an *unordered tree pattern matching*.

We may consider another problem, called an *ordered tree pattern matching*, defined below.

Definition 2 An 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) Preserve *left-to-right order*: For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if v_1 is to the left of v_2, then $f(v_1)$ is to the left of $f(v_2)$ in T.

In general, a node u_1 is said to be to the left of another node u_2 in a tree T if they are not related by the ancestor-descendant relationship and u_2 follows u_1 when we traverse T in preorder.

This kind of tree mappings is useful in practice. For example, an XML data model was proposed by Catherine and Bird [5] for representing interlinear text for linguistic applications, used to demonstrate various linguistic principles in different languages. For the purpose of linguistic analysis, it is essential to preserve the linear order between the words in a text [5]. In addition to interlinear text, the syntactic structure of textual data should be considered, which breaks a sentence into syntactic units such as noun clauses, verb phrases, adjectives, and so on. These are used by the language Tree-Bank [31] to provide a hierarchical representation of sentences. Therefore, by the evaluation of a tree pattern query against the TreeBank, the order between siblings should be considered [31, 34].

The remainder of the paper is organized as follows. In Section 2, we review the concept of XML data streams. In Section 3, we discuss our algorithm and analyze its computational complexities. The paper concludes in Section 4.

2 XML data stream

In a XML database, we can always store a document as a data stream by using an interesting tree encoding [46], which can be used to identify different relationships between the nodes of a tree.

Let T be a document tree. We associate each node v in T with a quadruple (d, l, r, ln), denoted as $\alpha(v)$, where d = DocId, l = LeftPos, r = RightPos, and ln = LevelNum, defined to be the nesting depth of the element in the document. (See Fig. 2 for illustration.) By using such a data structure, the structural relationships between the nodes in an XML database can be simply determined [46]:

(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) *left-to-right order*: 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 $d_1 = d_2$, $r_1 < l_2$.

In Fig. 4, v_2 is an ancestor of v_6 and we have v_2.LeftPos = 2 < v_6.LeftPos = 6 and v_2.RightPos = 9 > v_6.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.LeftPos = v.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 $d_1 = d_2$, $l_1 \leq l_2$, and $r_1 \geq r_2$, we say that α_2 is subsumed by α_1. For convenience, a quadruple is considered to be subsumed by itself (i.e., a node is considered to be an ancestor of itself). In this way, we can store an XML document as a stream of quadruples sorted by LeftPos or RightPos values.

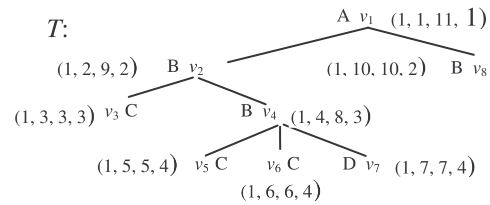

Fig. 2. Illustration for tree encoding

If no confusion is caused, we will used v and $\alpha(v)$ interchangeably. As with DeweyIDs [21], we can also leave gaps in the numbering space between consecutive labels to support dynamical changes of documents.

3 Algorithm

In this section, we discuss our strategy for the ordered tree pattern matching. First, we discuss the main algorithm in 3.1. Then, in 3.2, we show the correctness of our algorithm and analyze its computational complexities. In 3.3, we describe how to adapt it to an indexing environment, as well as how the wildcard and output node can be handled.

3.1 Algorithm description

Our algorithm works bottom-up. Therefore, we need to sort XML streams by (DocID, RightPos) values. Each time a query Q is submitted to the system, we will associate each query node q with a data stream $L(q)$ such that for each $v \in L(q)$ $label(v) = label(q)$, as illustrated in Fig. 3, in which each query node is attached with a list of matching nodes of the document tree shown in Fig. 2.

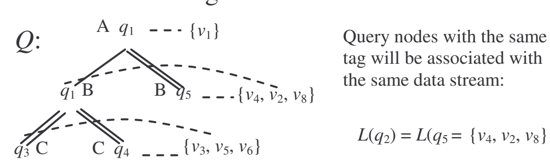

Fig. 3. Illustration for $L(q_i)$'s

In the figure, for simplicity, we use the node names in a data stream, instead of the nodes' quadruples. In addition, DocIDs are not displayed. We remark that the data streams associated with different nodes in Q may be the same. So we use $\boldsymbol{q}$ to represent the set of such query nodes and denote by $L(\boldsymbol{q})$ the data stream shared by them. Without loss of generality, assume that the query nodes in $\boldsymbol{q}$ are sorted by their RightPos values. We will also use $L(Q) = \{L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)\}$ to represent all the data streams with respect to Q, where each $\boldsymbol{q}_i$ $(i = 1, ..., l)$ is a set of sorted query nodes that share a common data stream.

In order to facilitate the checking of tree embedding, some more data structures are established for query nodes.

1. First, we will number the nodes of Q in postorder (see the boldfaced numbers in Fig. 4(a) for illustration). So the nodes in Q will be referenced by their postorder numbers.
2. For each node q of Q, a link from it to the left-most leaf node in $Q[q]$, denoted by $\delta(q)$, is established. (See Fig. 4(b)). For a leaf node q', $\delta(q') = q'$. Additionally, we set a *virtual node* for Q, numbered 0, which is considered to be to the left of any node in Q.

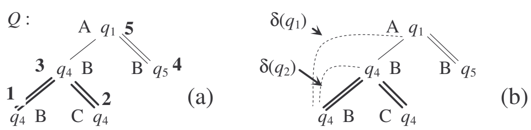

Fig. 4. Labeled trees and postorder numbering

3. Let q' be a leaf node in Q. We denote by $\delta^{-1}(q')$ a set of nodes $\boldsymbol{x}$ such that for each $q \in \boldsymbol{x}$ $\delta(q) = q'$.
4. Each time we create a node v in T', we associate it with an array A_v of length $|Q|$, indexed from 0 to $|Q|$ - 1. In A_v, each entry is a query node or ϕ, defined below:

$$A_v[q] = \begin{cases} \max\{x \mid x \in \delta^{-1}(q) \wedge T[v] \text{ embeds } Q[x], & \text{if there is a least leaf } q' \text{ larger than } q \text{ such that } \delta^{-1}(q) \text{ contains at least one node } x \text{ with } Q[x] \text{ being embedded in } T[v]; \\ \phi, & \text{otherwise.} \end{cases}$$

Here, $Q[x]$ represents a subtree of Q rooted at x.
See Fig. 5 for illustration.

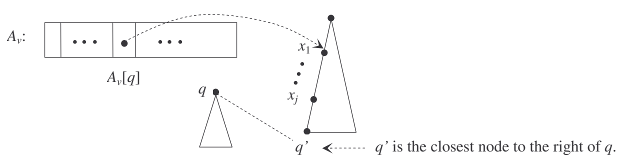

Fig. 5. Illustration for $A_v[q]$

In Fig. 5, q' represents a closest leaf node to the right of q (i.e., the least leaf node larger than q) such that there exists at least one $x \in \delta^{-1}(q')$ with $T[v]$ embedding $Q[x]$. Each x_i represents such a node in $\delta^{-1}(q')$. But we make $A_v[q]$ point to the largest one since the embedding of a tree in $T[v]$ implies the embedding of any of its subtrees in $T[v]$. At the same time, the left-to-right order is also recorded.
Such entries can be produced as below.

(i) If we find $Q[x]$ can be embedded in $T[v]$, we will set A_v $A_v[q_1]$, ..., $A_v[q_k]$ to x, where each q_l ($0 \le l \le k$) is a query node to the left of x, to record the fact that x is the closest node to the right of q_l such that $T[v]$ embeds $Q[x]$.
(ii) If some time later we find another node x', which is to the right of x, such that $Q[x']$ can be embedded in $T[v]$, we will set $A_v[p_1]$, ..., $A_v[p_s]$ to x', where each p_l ($1 \le l \le s$) is to the left of x' but to the right of q_k.
(iii) If x' is an ancestor of x, we will find all those entries pointing to a descendant of x' on the left-most path in $Q[x']$. Replace these entries with x'.
(iv) For all the other nodes v' such that $T[v']$ embeds $Q[x]$, we will set values for the entries in $A_{v'}$ in the same way as (i), (ii), and (iii).

As an example, consider node v_4 in T shown in Fig. 2. After it is checked against node 1 (q_3) of Q in Fig. 4, we will set A_{v_4} [0] to 1 since node 1 (q_3) of Q is the closest node to the right of node 0 (the virtual node of Q) such that $T[v_4]$ embeds $Q[q_3]$. (See Fig. 6(a)). At a later time point, we find that $T[v_4]$ also embeds $Q[q_2]$, we will change [0] to 3 (see Fig. 6(b)). It is because node 1 (q_3) is a descendant of node 3 (q_2) on the left-most path in $Q[q_2]$. In the subsequent computation, we will find that $T[v_4]$ can embed $Q[q_5]$. In order to record this fact, will be further modified as shown in Fig. 6(c) since node 4 (q_5) is the closest node right of node 1 (q_3), 2 (q_4), and 3 (q_2) such that $T[v_4]$ embeds $Q[q_5]$.

A_{v_4}: $[1, \phi, \phi, \phi, \phi]$ (a) $[3, \phi, \phi, \phi, \phi]$ (b) $[3, 4, 4, 4, \phi]$ (c)

Fig. 6. Changes in array A_v

Based on A_v's, the ordered tree embedding can be checked as follows:

- Let q in Q and v in T be the nodes encountered.
- Let v_1, ..., v_k be the child nodes of v. Let q_1, ..., q_l be the child nodes of q. We first check A_{v_1} starting from A_{v_1} $[l]$, where $l = \delta(q)$ - 1. We begin the searching from $\delta(q)$ - 1 because it is the closest node to the left of the first child of q. Let A_{v_1} $[l] = q'$. If q' is not an ancestor of q_1, we will check A_{v_2} $[l]$ in a next step. This process continues until one of the following conditions is satisfied:
  (i) All A_{v_i} 's have been checked, or
  (ii) There exists v_j such that $[l]$ is an ancestor of q.

If all A_{v_i} 's are checked (case (i)), it shows that $Q[q_1]$ cannot be embedded in any subtree rooted at a child node of v. So $T[v]$ cannot embed $Q[v]$.
If it is case (ii), we know that $T[v_j]$ embeds $Q[q_1]$. If q_1 is a //-child, or both q_1 and v_j are /-children, we will continue to check $[q_1]$. (Otherwise, we will continue to check A_{v_2} $[l]$.)

In terms of the above discussion, we give our algorithm for evaluating ordered tree pattern queries. It mainly consists of two processes: (1) scanning the data streams associated with the query nodes in such an order that each time the quadruple with the least RightPos is accessed; (2) checking tree embedding.

In the first process, for each encountered quadruple v from a $L(\boldsymbol{q})$, a node is created, which will be associated with two links, denoted respectively *left-sibling*(v) and *parent*(v), to reconstruct T (or a subtree of T, which contains only those nodes matching a query node) as follows:

1. Identify a data stream $L(\boldsymbol{q})$ with the first element being of the minimal RightPos value. Choose the first element v of $L(\boldsymbol{q})$. Remove v from $L(\boldsymbol{q})$.
2. Generate a node for v.
3. If v is not the first node, we do the following:
   Let v' be the node chosen just before v. If v' is not a child (descendant) of v, create a link from v to v', called a *left-sibling* link and denoted as *left-sibling*(v) = v'.
   If v' is a child (descendant) of v, we will first create a link from v' to v, called a *parent* link and denoted as *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 $parent(u) \leftarrow v$. Finally, set $left\text{-}sibling(v) \leftarrow v''$.

In the second process, we check, for each v from a $L(\boldsymbol{q})$, whether $T[v]$ embeds $Q[q]$ for each q in $\boldsymbol{q}$. For this purpose, in addition to $\delta(q)$, $\boldsymbol{A}_v$, another two data structures are used:

S_v - a list of query nodes q such that $T[v]$ embeds $Q[q]$.

$\tau(v)$ - a postorder number for a query node associated with node v in T' such that $T'[v]$ is the subtree currently found to embed $Q[\tau(v)]$. The initial value for each $\tau(v)$ is 0.

In the following algorithm, we only show the second process for ease explanation. But the first process can easily be integrated.

Algorithm *tree-embedding*($L(Q)$)
Input: all data streams $L(Q)$.
Output: S_v's, which show the tree embedding.
begin
1. **repeat until** each $L(\boldsymbol{q})$ in $L(Q)$ become empty
2. {identify $\boldsymbol{q}$ such that the first element v of $L(\boldsymbol{q})$ is of the minimal RightPos value; remove v from $L(\boldsymbol{q})$;
3. generate node v; $A_v \leftarrow \phi$; $S_v \leftarrow \phi$;
4. let $v_1, ..., v_k$ be the children of v.
5. **for** each $q \in \boldsymbol{q}$ **do** { (*nodes in $\boldsymbol{q}$ are sorted.*)
6. let $q_1, ..., q_l$ be the children of q;
7. **if** $l = 0$ **then** $j \leftarrow 0$
8. **else** { $p \leftarrow \delta(q) - 1$;
9. $i \leftarrow 1$; $j \leftarrow 1$; $p' \leftarrow A_{v_i}[p]$;
10. **while** $i \le k$ and $p' \ne \phi$ and $p' < q$ **do**
11. {**if** (p' is an ancestor of q_j and ((q, q_j) is a //-edge, or both (q, q_j) and (v, v_i) are /-edges))
12. **then** { $p' \leftarrow A_{v_{i+1}}[p']$; $i \leftarrow i + 1$; $j \leftarrow j + 1$;}
13. **else** {$p' \leftarrow A_{v_{i+1}}[p]$; $i \leftarrow i + 1$;}
14. }
15. }
16. **if** $l = j$ **then**
17. { $S_v \leftarrow S_v \cup \{q\}$;
18. **if** q is to the right of $\tau(v)$
19. **then** { $a \leftarrow \tau(v)$;
20. **for** $b = a$ to $q - 1$ **do**
21. {**if** b is to the left of q **then** $A_v[b] \leftarrow q$;}
22. }
23. **else** {replace with q all those entries pointing to a descendant of q on the left-most path in $Q[q]$ in A_v; }
24. $\tau(v) \leftarrow q$; }
25. **for** $i = 1$ to k **do** {$A_v \leftarrow merge(A_v, A_{v_i})$;}
26. remove $A_{v_1}, ..., A_{v_k}$;
27. }
end

In the above algorithm, the nodes in T are created one by one. But for each node v generated for an element from a $L(\boldsymbol{q})$, A_v is created and each entry is initialized to ϕ. Then, for each $q \in \boldsymbol{q}$, we will check whether $T[v]$ embeds $Q[q]$. This is done by executing lines 7 - 15, in which two index variables: i and j are used to scan the children of v and q, respectively. The searching begins from [p], where $p = \delta(q) - 1$ (see line 8). In each iteration of the **while**-loop (see lines 10 - 14), we check v_i against q_j by examining whether the following two conditions are satisfied:

i) $A_{v_i}[p]$ is an ancestor of q_j, and

ii) (q, q_j) is a //-edge, or both (q, q_j) and (v, v_i) are /-edges.

If both the conditions hold, $T[v_i]$ embeds $Q[q_j]$. We will continue to check $T[v_{i+1}]$ against $Q[q_{j+1}]$. Special attention should be paid to the statement: $p' \leftarrow A_{v_{i+1}}[p']$ (line 12), by which we get a query node q' that is the closest to the right of q_j, such that $T[v_{i+1}]$ embeds $Q[q']$. We also notice that if $T[v_i]$ cannot embed $Q[q_j]$, we will check v_{i+1} against q_j by doing $p' \leftarrow A_{v_{i+1}}[p]$ (see line 13).

This process continues until one of the following conditions is met: (1) $i > k$, (2) $p' = \phi$, or (3) $p' \ge q$. (1) or (2) implies an unsuccessful checking. If (3) holds, we must have $l = j$ (see line 16), showing that each $Q[q_{j'}]$ ($1 \le j' \le l$) is embedded by a $T[v_i]$ ($1 \le i \le k$) in the left-to-right order.

Lines 18 - 23 are used to set the entries in A_v.

Finally, we need to merge each A_{v_i} into A_v (line 25) since the embedding of a subtree in $T[v_i]$ implies the embedding of that subtree in $T[v]$.

Handling ϕ as a negative integer (e.g., -1) that represents a descendant of any node, we define $merge(A_v, A_{v_i})$ as below:

$$merge(A_v, A_{v_i})[j] = \begin{cases} \max(A_v[j], A_{v_i}[j]), & \text{if } A_v[j] \text{ and } A_{v_i}[j]) \text{ are on the same path;} \\ \max(A_v[j], A_{v_i}[j]), & \text{otherwise.} \end{cases}$$

Obviously, if $A_v[j]$ and $A_{v_i}[j]$ are on the same path, $merge(A_v[j], A_{v_i}[j])$ should be set to be $max\{A_v[j], A_{v_i}[j]\}$. However, if $A_v[j]$ and $A_{v_i}[j]$ are on different paths, $merge(A_v[j], A_{v_i}[j])$ is set to be $min\{A_v[j], A_{v_i}[j]\}$. It is because in A_v each entry $A_v[j]$ is the closest node j' to the right of j such that $T[v]$ contains $Q[j']$.

In line 26, we remove $A_{v_1}, ..., A_{v_k}$ since they will not be used any more.

Example 2 As an example, consider Q shown in Fig. 3 and T in Fig. 3 once again. The nodes in Q are postorder numbered, i.e., $q_1 = 5$, $q_2 = 3$, $q_3 = 1$, $q_4 = 2$, and $q_5 = 4$. When we apply the above algorithm to them, each node v (except v_7) in T will be associated with an array A_v as shown in Fig. 6.

In Step 1, v_3 is checked against $\boldsymbol{q}'' = \{2\}$. Node 2 is a leaf node. So, we have $T[v_3]$ embedding $Q[2]$ and A_{v_3} will be established as shown in Fig. 6(a). We notice that $A_{v_3}[0] = A_{v_3}[1] = 2$. It is because node 2 is the closest node to the right of node 0 (virtual node) and node 1 (q_3) such that $T[v_3]$ embeds $Q[2]$.

In Setp 2, v_5 is checked against $\boldsymbol{q}' = \{1, 3, 4\}$. Node 1 is a leaf and so $T[v_5]$ embeds $Q[1]$, which sets A_{v_5} as shown in Fig. 6(b). $T[v_5]$ is not able to contain $Q[3]$, but $Q[4]$. Thus, A_{v_5} is changed to $[1, 4, 4, 4, \phi]$.

In Setp 3, v_6 is checked against $\boldsymbol{q}'' = \{2\}$. Node 2 is a leaf and $T[v_6]$ embeds $Q[2]$. A_{v_6} is the same as . See Fig. 6(c).

In Setp 4, v_4 is checked against $\boldsymbol{q}' = \{1, 3, 4\}$. Since $T[v_4]$ embeds $Q[1]$, A_{v_4} is first set to $[1, \phi, \phi, \phi, \phi]$ (see Fig. 6(d)). When v_4 is checked against node 3, their children will be examined. The children of v_4 are v_5 and v_6; and the children of node 3 are nodes 1 and 2. First, $A_{v_5}[0]$ is checked. It is 1, showing that $T[v_5]$ embeds $Q[1]$. Next, $A_{v_6}[1]$ is checked, it is equal to 2, showing that $T[v_6]$ embeds $Q[2]$. Therefore, $T[v_4]$ is able to embed $Q[3]$ and A_{v_4} is changed to $[3, \phi, \phi, \phi, \phi]$. (Note that node 1 is a child of node 3 and also on the left-most path in $Q[3]$.) By checking v_4 against node 4, A_{v_4} becomes $[3, 4, 4, 4, \phi]$. By *merge*(A_{v_4}, A_{v_5}), A_{v_4} is not changed. But by *merge*(A_{v_4}, A_{v_6}), A_{v_4} is further changed to $[3, 2, 4, 4, \phi]$.

A_{v_3}: $[2, 2, \phi, \phi, \phi]$ (a)
A_{v_5}: $[1, \phi, \phi, \phi, \phi] \Rightarrow [1, 4, 4, 4, \phi]$ (b)
A_{v_6}: $[2, 2, \phi, \phi, \phi]$ (c)
A_{v_4}: $[1, \phi, \phi, \phi, \phi] \Rightarrow [3, \phi, \phi, \phi, \phi] \Rightarrow [3, 4, 4, 4, \phi] \Rightarrow [3, 2, 4, 4, \phi]$ (d)
A_{v_2}: $[1, \phi, \phi, \phi, \phi] \Rightarrow [3, 2, \phi, \phi, \phi] \Rightarrow [3, 2, 4, 4, \phi]$ (e)
A_{v_8}: $[1, 4, 4, 4, \phi]$ (f)
A_{v_1}: $[5, \phi, \phi, \phi, \phi] \Rightarrow [5, 2, 4, 4, \phi]$ (g)

Fig. 6. A sample trace

The same analysis applies to Step 5 and 6, by which and are constructed as shown in Fig. 6(e) and (f), respectively.

In Step 7, v_1 is checked against $\boldsymbol{q} = \{5\}$. We will first check their children. The children of v_1 are v_2 and v_8; and the children of node 5 are nodes 3 and 4. Since $A_{v_2}[0] = 3$, showing that $T[v_2]$ contains $Q[3]$. However, since the edge (5, 3) (i.e., (q_1, q_2)) in Q is a /-edge, we have to check whether v_2 in T is a /-child of v_1. It is the case. So we will continue to check $A_{v_8}[3]$. It is equal to 4, demonstrating that $T[v_8]$ embeds $Q[4]$. Thus, A_{v_1} is set to $[5, \phi, \phi, \phi, \phi]$. See Fig. 6(g). By *merge*(A_{v_1}, A_{v_2}), A_{v_1} is changed to $[5, 2, 4, 4, \phi]$. By *merge*(A_{v_1}, A_{v_8}), A_{v_1} remains unchanged.

3.2 Corectness and time complexities

In this subsection, we prove the correctness of the algorithm and analyze its computational complexities.

Proposition 2 Algorithm *tree-matching*( ) computes the values in A_v's correctly.

Proof. We prove the proposition by induction on the heights of nodes in T'. We use $h(v)$ to represent the height of node v.

Basic step. It is clear that any node v with $h(v) = 0$ is a leaf node. Then, each entry in A_v corresponds to a leaf node q in Q with *label*(v) = *label*(q). Since all those leaf nodes in Q are checked in the order of increasing RightPos values, the entries in A_v must be correctly established.

Induction step. Assume that for any node v with $h(v) \leq l$, the proposition holds. We will check any node v with $h(v) = l + 1$.

Let $v_1, ..., v_k$ be the children of v. Then, for each v_i ($i = 1, ..., k$), we have $h(v_i) \leq l$. In terms of the induction hypothesis, each A_{v_i} is correctly constructed. Let $q_1, ..., q_l$ be the children of q. In the main **while**-loop, we will access a sequence:

$A_{v_1}[p_1], ..., A_{v_k}[p_k]$

with $p_1 \leq p_2 \leq ... \leq p_k$. If there exists a subsequence: p_{i_1}, ..., p_{i_l} satisfying the following conditions:

i) p_{i_j} is an ancestor of q_j, and

ii) (q, q_j) is a //-edge, or both (q, q_j) and $(v,)$ are /-edges,

$T[v]$ embeds $Q[q]$. If q is to the right of $\tau(v)$, then in A_v, all the entries to the right of $\tau(v)$ but to the left of q will be set to be q. If q is an ancestor, all those entries pointing to a descendant of q and appearing on the left-most path in $Q[q]$ are replaced with q. The merging operation is obviously correct since the embedding of a subtree in $T[v_i]$ implies that $T[v]$ also contains that subtree. This completes the proof.

Now we analyze the time complexity of the algorithm. The whole cost can be divided into four parts.

The first part consists of checking v of T' against q of Q. Since in each A_{v_i}, where v_i is a child of v, only one entry is checked, this part of cost is bounded by

$$O(\sum_{v \in T} d_v) = O(|T|),$$

where d_v represents the outdegree of v.

The second part is the cost for filling in the case that q is to the right of $\tau(v)$. For each $v \in T$, the cost is bounded by $O(|Q|)$. So this part of cost is in the order of $O(|T|\cdot|Q|)$.

The third part is the cost for filling in the case that q is an ancestor of $\tau(v)$. This part of checking can be slightly improved as follows. In A_v, each entry is set to be a pointer to a place storing a postorder number, instead of the number itself, as illustrated in Fig. 7.

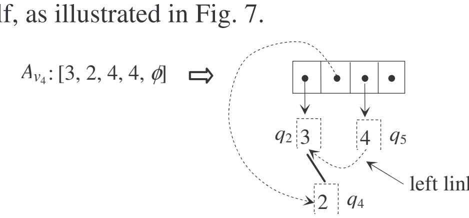

Fig. 7. Structure of A_v

In Fig. 7, A_{v_4} is stored as an array of pointers. Especially, the postorder numbers in A_v a can be organized as a tree in a way similar to the reconstruction of T from data streams. Therefore, to modify all the entries pointing to a descendant of q on the left-most path in $Q[q]$, we need only to search for the place containing the corresponding postorder number. This can be done by traversing along a left-link chain as discussed in 3.1. For a node $v \in T$ checked against $\boldsymbol{q}$, the cost of this process is bounded by $O(|\boldsymbol{q}|)$.

The forth part of cost is for the merging operation. It can simply be estimated by

$O(= O(|T|\cdot|Q|)$.

In terms of the above analysis, we have the following proposition.

Proposition 3 The time complexity of Algorithm *tree-embedding*( ) is bounded by O(|T'|·|Q| + O(|D|·|Q|).

The space overhead of Algorithm *tree-embedding*( ) is in the order of O($leaf_T$·|Q|), where $leaf_T$ is the number of the leaf nodes of T. It is because after a v is checked all the arrays associate with its children are removed. So at any time point during the execution, at most $leaf_T$ nodes in T are associated with a array (see line 26 in Algorithm *tree-embedding*( ).)

3.3 About index, *, and output nodes

In the previous subsections, the main algorithm has been described in detail. However, three issues yet remain to be addressed. That is, the indexing, wildcards (*) as well as the output node in Q should be handled carefully.

\- *Index*

The index mechanism used in our implementation is a modified XB-tree [4]. As with *TwigStack* [4], an XB-tree is established over a data stream sorted by LeftPos values. However, we can use the following algorithm to make a transformation of data streams, in which a global stack ST is maintained to control the process. In ST, each entry is a pair (q_i, v) with $q_i \in Q$ and $v \in T$ (v is represented by its quadruple.)

Algorithm *stream-transformation*($B(q_i)$'s)

input: all data streams $B(q_i)$'s, each sorted by LeftPos.

output: new data streams $L(q_i)$'s, each sorted by RightPos.

begin

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

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 $L(q_i)$'s (see lines 3 - 5); and push v into ST (see line 6.) The output of the algorithm is a set of data streams $L(q_i)$'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 $L(q_i)$'s.

In Fig. 8(b), we demonstrate a XB-tree built on a $B(\boldsymbol{q})$ shown in Fig. 8(a).

Each entry in a page (a node) P of an XB-tree consists of a bounding segment [LeftPos, RightPos] and a pointer to its child page, which contains entries with bounding segments completely included in [LeftPos, RightPos]. The bounding segments may partially overlap, but their LeftPos positions are in increasing order. Besides, each page 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 ith pointer in P.*parent* points to P. For instance, in the XB-tree shown in Fig. 8(b), P_3.*parentIndex* = 2 since the second pointer in P_1 (the parent of P_3) points to P_3.

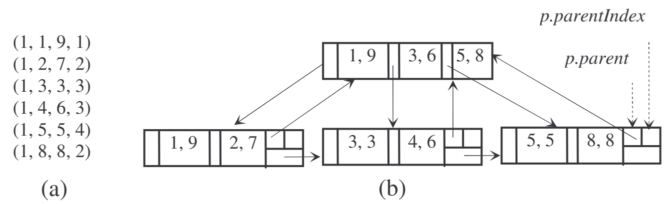

Fig. 8. A quadruple sequence and the XB-=tree over it

In our implementation, some modifications have been made. First, for a set of nodes $\boldsymbol{q} = \{q_1, ..., q_l\}$, we establish only one XB-tree, where q_1, ..., q_l have the same label. But for each $q_j \in \mathbf{q}$ (j = 1, ..., l), we maintain a pair (P, i), denoted , to indicate that the ith entry in the page P (in the XB-tree) is currently accessed for q_j. Thus, each (j = 1, ..., l) corresponds to a different searching of the same XB-tree as if we have a separate copy of that XB-tree over $B(q_j)$. We use *advance*() and *drilldown*() to navigate the corresponding XB-tree. Concretely, *advance*() advances i. If i is the last entry in P, is replaced with (P.*parent*, P.*parentIndex*). By *drilldown*(), we replace (P, i) with (P', 0) if P is not a leaf page, where P' is the child page pointed to by the pointer of the ith entry in P [4].

The second modification consists in a different navigation strategy of XB-trees. By *Twigstack* [4], each time to 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_q in $B(q)$ such that it has a descendant in each of the streams $B(q_i)$ (where q_i is a child of q.)

ii) Each recursively satisfies (i).

iii) LeftPos(v_q) is minimum.

But for the ordered tree matching, (i) is changed:

- For q, there exists an entry v_q in $B(q)$ such that it has a descendant in each of the streams $B(q_i)$. If q has a right sibling q', then there exists an entry $v_{q'}$ in $B(q')$, which is to the right of v_q.

In this way, not only the *ancestor-descendant* relationship, but also the *left-to-right* order is utilized to skip over entries in an XB-tree, which substantially reduces the number of disk access.

\- *Wildcards*

Using XB-trees, * is handled in the same way as non-wildcard nodes. In fact, for each q in Q, no matter whether it is a wildcard or not, we will be looking for only one element in the corresponding XB-tree each time. More importantly, using the above *drilldown* and *advance* operators [4], any entry in an XB-tree (corresponding to a query node) is accessed only once.

- *Output node*

As for the output node of Q, we should notice that the set S_v generated for each node v in T' does not serve as the answer to Q although for each $q \in S_v$ we have $T'[v]$ embeds $Q[q]$. For this reason, we need to slightly modify the algorithm to create two extra data structures L_r and L_o as below.

Each time we insert a q into an S_v (see line 17 in Algorithm *tree-embedding*( )), we will also add v to L_r if q is the root r of Q, or to L_o if q is the output node o.

Clearly, in these two data structures, all nodes are increasingly sorted by the RightPos values. Thus, using them, we can create another subtree T'' of T (in a way similar to the generation of a matching subtree; see Algorithm *matching-tree-construction*( )). It contains only those nodes v such that $T'[v]$ embeds $Q[r]$ with *label*(v) = *label*(r) or embeds $Q[o]$ with *label*(v) = *label*(o). We call a node v an *r-node* if $T'[v]$ contains $Q[r]$ with *label*(v) = *label*(r), or an *o-node* if $T'[v]$ embeds $Q[o]$ with *label*(v) = *label*(o). Search T''. Any node v, which is an o-node and also a child of some r-node, should be an answer if o is a descendant of r or a //-child of r. If o is a /-child of r, an o-node is an answer only if it is a /-child of some r-node.

4 Conclusion

In this paper, a new algorithms *tree-embedding* for processing ordered tree pattern queries is discussed. For the ordered tree pattern queries, not only the parent-child and ancestor-descendant relationships but also the order of siblings are taken into account. The time complexity of the algorithm is bounded by $O(|T|\cdot|Q|)$ and its space overhead is by $O(leaf_T\cdot|Q|)$, where Q stands for a tree pattern, T' for a subtree of a document tree T containing the nodes that match at least one query node, and $leaf_T$ represents the number of the leaf nodes of T'. Our experiments demonstrate that our method is both effective and efficient for the evaluation of ordered tree pattern queries.

5 References

[1] R. Agrawal, A. Borgida and H.V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," *Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data*, Oregon, 1989, pp. 253-262.

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

[2] A. Aghili, H. Li, D. Agrawal, and A.E. Abbadi, TWIX: Twig structure and content matching of selective queries using binary labeling, in: *INFOSCALE*, 2006.

[3] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu, Structural Joins: A primitive for efficient XML query pattern matching, in *Proc. of IEEE Int. Conf. on Data Engineering*, 2002.

[4] N. Bruno, N. Koudas, and D. Srivastava, Holistic Twig Joins: Optimal XML Pattern Matching, in *Proc. SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 2002, pp. 310-321.

[5] B. Catherine and S. Bird, Towards a general model of Interlinear text, in *Proc. of EMELD Workshop*, Lansing, MI, 2003.

[6] D. D. Chamberlin, J.Clark, D. Florescu and M. Stefanescu. "XQuery1.0: An XML Query Language," http:// www.w3.org/TR/query-datamodel/.

[7] D. D. Chamberlin, J. Robie and D. Florescu. "Quilt: An XML Query Language for Heterogeneous Data Sources," *WebDB 2000*.

[8] T. Chen, J. Lu, and T.W. Ling, On Boosting Holism in XML Twig Pattern Matching, in: *Proc. SIGMOD*, 2005, pp. 455-466.

[9] Y. Chen, An Efficient Streaming Algorithm for Evaluating XPath Queries. in *Proc. WEBIST*, 2008, pp. 190-196.

[10] B. Choi, M. Mahoui, and D. Wood, On the optimality of holistic algorithms for twig queries, in: *Proc. DEXA*, 2003, pp. 235-244.

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

[12] Y. Chen, S.B. Davison, Y. Zheng, An Efficient XPath Query Processor for XML Streams, in *Proc. ICDE*, Atlanta, USA, April 3-8, 2006.

[13] S. Chen, H-G. Li, J. Tatemura, W-P. Hsiung, D. Agrawa, and K.S. Canda, *Twig^{2}Stack*: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents, in *Proc. VLDB*, Seoul, Korea, Sept. 2006, pp. 283-294.

[14] B.F. Cooper, N. Sample, M. Franklin, A.B. Hialtason, and M. Shadmon, A fast index for semistructured data, in: *Proc. VLDB*, Sept. 2001, pp. 341-350.

[15] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D.Suciu, A Query Language for XML, in: *Proc. 8th World Wide Web Conf.*, May 1999, pp. 77-91.

[16] D. Florescu and D. Kossman, Storing and Querying XML data using an RDMBS, *IEEE Data Engineering Bulletin*, 22(3):27-34, 1999.

[17] G. Gou and R. Chirkova, Efficient Algorithms for Evaluating XPath over Streams, in: *Proc. SIGMOD*, June 12-14, 2007.

[18] R. Goldman and J. Widom, DataGuide: Enable query formulation and optimization in semistructured databases, in: *Proc. VLDB*, Aug. 1997, pp. 436-445.

[19] G. Gottlob, C. Koch, and R. Pichler, Efficient Algorithms for Processing XPath Queries, *ACM Transaction on Database Systems*, Vol. 30, No. 2, June 2005, pp. 444-491.

[20] C.M. Hoffmann and M.J. O'Donnell, Pattern matching in trees, *J. ACM*, 29(1):68-95, 1982.

[21] Sayyed Kamyar Izadi a, Theo Härder b,*, Mostafa S. Haghjoo, S^3: Evaluation of Tree-Pattern Queries Supported by Structural Summaries, Data & Knowledge Engineering, 68, pp. 126-145, Elsevier, Sept. 2008.

[22] R.B. Lyngs, M. Zuker & C.N.S. Pedersen, Internal loops in RNA secondary structure prediction, in *Proceedings of the 3rd annual international conference on computational molecular biology (RECOMB)*, 260-267 (1999).

[23] Jiang, Z., Luo, C., Hou, W.-C., Zhu, Q., and Che, D., “Efficient Processing of XML Twig Pattern: A Novel One-Phase Holistic Solution,” In *Proc. the 18th Int’l Conf. on Database and Expert Systems Applications (DEXA)*, pp. 87-97, Sept. 2007.

[24] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth, Covering indexes for branching path queries, in: *ACM SIGMOD*, June 2002.

[25] C. Koch, Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach, in: *Proc. VLDB*, Sept. 2003.

[26] Q. Li and B. Moon, Indexing and Querying XML data for regular path expressions, in: *Proc. VLDB*, Sept. 2001, pp. 361-370.

[27] Y. Rui, T.S. Huang, and S. Mehrotra, Constructing table-of-content for videos, *ACM Multimedia Systems Journal, Special Issue Multimedia Systems on Video Libraries*, 7(5):359-368, Sept 1999.

[28] J. Lu, T.W. Ling, C.Y. Chan, and T. Chan, From Region Encoding to Extended Dewey: on Efficient Processing of XML Twig Pattern Matching, in: *Proc. VLDB*, pp. 193 - 204, 2005.

[29] J. McHugh, J. Widom, Query optimization for XML, in *Proc. of VLDB*, 1999.

[30] G. Miklau and D. Suciu, Containment and Equivalence of a Fragment of XPath, *J. ACM*, 51(1):2-45, 2004.

[31] K. Müller, Semi-automatic construction of a question treebank, in *Proc. of the 4th Intl. Conf. on Language Resources and Evaluation*, Lisbon, Portual, 2004.

[32] L. Qin, J.X. Yu, and B. Ding, “TwigList: Make Twig Pattern Matching Fast,” In Proc. 12th Int’l Conf. on Database Systems for Advanced Applications (DASFAA), pp. 850-862, Apr. 2007.

[33] P. Ramanan, Holistic Join for Generalized Tree Patterns, *Information Systems* 32 (2007) 1018-1036.

[34] P. Rao and B. Moon, Sequencing XML Data and Query Twigs for Fast Pattern Matching, *ACM Transaction on Database Systems*, Vol. 31, No. 1, March 2006, pp. 299-345.

[35] A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse, The XML benchmark project, Technical Report INS-Ro1o3, Centrum voor Wiskunde en Informatica, 2001.

[36] C. Seo, S. Lee, and H. Kim, An Efficient Index Technique for XML Documents Using RDBMS, *Information and Software Technology* 45(2003) 11-22, Elsevier Science B.V.

[37] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. Dewitt, and J.F. Naughton, Relational databases for querying XML documents: Limitations and opportunities, in *Proc. of VLDB*, 1999.

[38] U. of Washington, The Tukwila System, available from http:/ /data.cs.washington.edu/integration/tukwila/.

[39] U. of Wisconsin, The Niagara System, available from http:// www.cs.wisc.edu/niagara/.

[40] U of Washington XML Repository, available from http:// www.cs.washington.edu/research/xmldatasets.

[41] H. Wang, S. Park, W. Fan, and P.S. Yu, ViST: A Dynamic Index Method for Querying XML Data by Tree Structures, *SIGMOD Int. Conf. on Management of Data*, San Diego, CA., June 2003.

[42] H. Wang and X. Meng, On the Sequencing of Tree Structures for XML Indexing, in *Proc. Conf. Data Engineering*, Tokyo, Japan, April, 2005, pp. 372-385.

[43] World Wide Web Consortium. XML Path Language (XPath), W3C Recommendation, 2007. See http://www.w3.org/TR/ xpath20.

[44] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, Jan. 2007. See http://www.w3.org/TR/xquery.

[45] XMARK: The XML-benchmark project, http://monet-db.cwi.nl/xml, 2002.

[46] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman, on Supporting containment queries in relational database management systems, in *Proc. of ACM SIGMOD*, 2001.

[47] M. Götz, C. Koch, and W. Martens, Efficient Algorithms for the tree Homeomorphism Problem, in *Pro. Int. Symposium on Database Programming Language*, 2007.

[48] Z. Bar-Yossef, M. Fontoura, and V. Josifovski, On the memmory requirements of XPath evaluation over XML streams, *Journal of Computer and System Sciences* 73 (2007) 391-441.

[49] M. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of KDD*, 2002.