Subtree Reconstruction, Query Node Intervals and Tree Pattern Query Evaluation*

YANGJUN CHEN AND YIBIN CHEN
Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada R3b 2E9
E-mail: y.chen@uwinnipeg; chenyibin@gmail.com

Since the extensible markup language XML emerged as a new standard for information representation and exchange on the Internet, the problem of storing, indexing, and querying XML documents has been among the major issues of database research. In this paper, we study the tree pattern matching and discuss a new algorithm for processing *ordered tree pattern queries*, by which not only ancestor/descendant relationships, but also left-to-right ordering of query nodes are considered. Such kind of tree matching has many applications in practice, such as the linguistic analysis, the video content-based retrieval, as well as the computational biology and the data mining. The time complexities of the new algorithm is bounded by $O(|D| \cdot |Q| + |T| \cdot leaf_Q)$ and its space overhead is by $O(leaf_T \cdot leaf_Q)$, where T stands for a document tree, Q for a tree pattern and D is the largest data stream among all the data streams associated with the nodes in Q. Each data stream contains the database nodes that match the predicate at a node q. $leaf_T$ ($leaf_Q$) represents the number of the leaf nodes of T (resp. Q). In addition, the algorithm can be adapted to an indexing environment with *XB-trees* being used. Experiments have been conducted, which shows that our algorithm is promising.

Keywords: XML documents, tree pattern queries, tree matching, tree encoding, XB-trees

1. INTRODUCTION

In XML [39, 40], data are represented as a tree; associated with each node of the tree is an element name from a finite alphabet $\sum$. 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 [39], XQuery [40], XML-QL [14], 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 $\sum \cup \{*\}$ (* is a wild-card, 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, 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 an ancestor of some node of name *f* (node 5). The query corresponds to the following XPath expression:

Received June 5, 2010; revised April 5, 2011; accepted August 1, 2011.
Communicated by Vincent S. Tseng.
* Results of this paper were partially presented at 20th International Conference on Database and Expert Systems Applications, 2009.

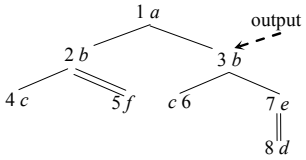


Fig. 1. A query tree.

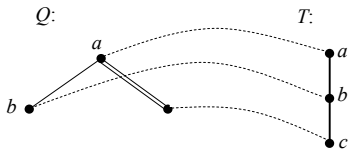


Fig. 2. A tree matching a path.

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

In Fig. 1, 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.

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 ($\in \sum \cup \{*\}$) 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:

(1) Preserve node label: For each $u \in Q$, *label*(u) = *label*($f(u)$) (or say, u matches $f(u)$).
(2) 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 embedded into T, or say, T contains Q.

Almost all the existing strategies for evaluating twig join patterns are designed according to this definition [4, 8-13, 21, 23-27, 32, 33, 42].

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

It is because by Definition 1 the left-to-right relationships among nodes 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, for which the left-to-right order is significant.

First, we define the sets of *left* and *right relatives* of a node v (denoted as $Lr(v)$ and $Rr(v)$, respectively).

Definition 2 Let T be a document tree. Let V be the set of its nodes and v be a node in T. The set of left relatives of v is defined by

$Lr(v) = \{u \in V \mid u$ and v are not related by ancestor/descendant or parent/child relationship, and v follows u when T is traversed in preorder.$\}$ ❑

See Fig. 3 for illustration.

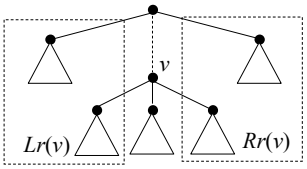


Fig. 3. Illustration for left and right relatives.

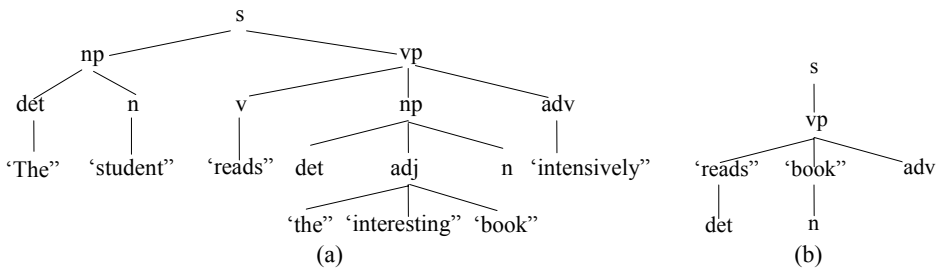


Fig. 4. A target tree and a pattern tree.

In a similar way, we can define $Rr(v)$.

Definition 3 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:

(1) same as (1) in Definition 1.
(2) same as (2) in Definition 1.
(3) Preserve *left-to-right order*: For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if v_1 is in $Lr(v_2)$, then $f(v_1)$ is in $Lr(f(v_2))$ in T. ❑

v_1 is said to be to the left of v_2 if v_1 is in $Lr(v_2)$. (Note that v_1 and v_2 can be siblings or in different subtrees.)

This kind of tree mappings is useful in practice. For instance, 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 TreeBank [27] 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 [27, 30]. As an example, consider the parse tree of a natural language sentence, shown in Fig. 4 (a).

One might want to locate, say, those sentences that include a verb phrase containing the verb "reads" and after it a noun "book" followed by any adverb. Such a query can also be represented as a tree, as shown in Fig. 4 (b).

Therefore, the evaluation of such a query is in fact a tree matching problem, by which both the ancestor/descendant relationship and the left-to-right ordering need to be taken into account.

Another application of the ordered tree matching is the video content-based retrieval. According to Rui *et al*. [46], a video can be successfully decomposed into a hierarchical tree structure, in which each node represents a scene, a group, a shot, a frame, a feature, and so on. Especially, such a tree is an ordered one since the temporal order is very important for video.

In addition, ordered tree matching can also be applied in the scene analysis, the computational biology (such as *RNA* structure matching [45]), as well as in the data mining (such as tree mining [47]).

In 2003, Wang *et al*. [37] proposed a first index-based method, called *ViST*, for handling ordered tree pattern queries, by which the XML data are transformed into structure-encoded sequences and stored in a disk-based virtual *trie* using B^+-trees. One of the problems of this method is that the query processing strategy by straightforward sequence matching may result in false alarms. Another problem, as pointed out in [30], the size of indexes is higher than linear in the total number of elements in an XML document. Such problems are removed by a method, called *PRIX*, discussed in [30]. This method constructs two Prüfer sequences to represent an XML document: a numbered Prüfer sequence and a labeled Prüfer sequence. For all the labeled Prüfer sequences, a virtual trie is constructed, used as an index structure. In this way, the size of indexes is dramatically reduced to $O(|T|)$. But it suffers from very high *CPU* time overhead according to the following analysis. The method consists of a string matching phase and several so-called refinement phases, for which $O(k|Q|\log|Q|)$ time is needed (see page 328 in [30]), where k is the number of subsequences of a labeled Prüfer document sequence, which match Q's labeled Prüfer sequence. However, by the string matching defined in [30], a query pattern string can match non-consecutive segments within a document target string (see Definition 4.1 in [30], page 306). So in the worst case k is in the order of $O(|T|^{|Q|})$ since for each position i (in the target) matching the first element in the pattern string the second element of the pattern can match possibly at $|T| - i - 1$ positions; and for each position j matching the second element in the pattern, the third element in the pattern can possibly match at $|T| - j - 1$ positions, and so on. As an example, consider the following Prüfer string:

$$a\ldots ab\ldots bc\ldots cd\ldots d,$$

in which each substring containing the same characters is of length $n/4$. Assume that the Prüfer string for a query is *abcd.* Then, there are $O(n^4)$ matching positions. For each of them, a tree embedding will be examined. (We note that if the string matching is restricted to consecutive segments, there is at most one matching for each position, at which the first element in the pattern matches. But it is not the case discussed in [30]. So, *PRIX* is an exponential time algorithm.)

In this paper, we propose a new method for processing ordered tree pattern queries. As in [30], we store documents as sequences, but each sequence contains only those nodes having the same label. In addition, in a sequence, all the nodes are represented by a kind of tree encoding such that their positions can be recognized. (We refer to these sequences as *data streams*.) Then, we design an algorithm for reconstructing subtree structures from the data streams, and a new tree labeling technique for query trees to represent left-to-right relationships, which enables us to efficiently check different relationships between the nodes. The new algorithm runs in $O(|D| \cdot |Q| + |T| \cdot leaf_Q)$ time and $O(leaf_T \cdot leaf_Q)$ space,

where $leaf_T$ ($leaf_Q$) represents the number of the leaf nodes of T (resp. Q), and D is the largest data stream among all the data streams associated with the nodes of Q.

The remainder of the paper is organized as follows. In section 2, we restate the tree encoding [42], which can be used to facilitate the recognition of different relationships among the nodes of trees. In section 3, we discuss our algorithm for evaluating ordered tree pattern queries. In section 4, we show how the XB-tree mechanism [4] can be integrated into it to speed up disk access. Section 5 is devoted to experiments. In section 6, we review the related work. Finally, a short conclusion is set forth in section 7.

2. TREE ENCODING

In [42], a tree encoding method was discussed, which is in fact the well known concept of time stamps produced during a depth-first search of a tree, and 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 (*DocId*, *LeftPos*, *RightPos*, *LevelNum*), denoted as $\alpha(v)$, 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 the element, respectively; and LevelNum is the nesting depth of the element in the document. (See Fig. 5 for illustration.) By using such a data structure, the structural relationships between the nodes in an XML database can be simply determined:

(1) *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 $r_1 > r_2$.
(2) *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 $ln_2 = ln_1 + 1$.
(3) *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 $d_1 = d_2$, $r_1 < l_2$.

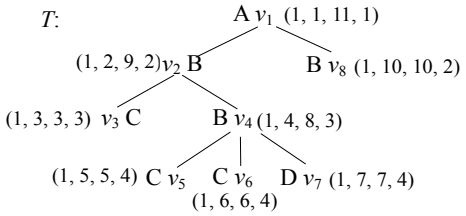
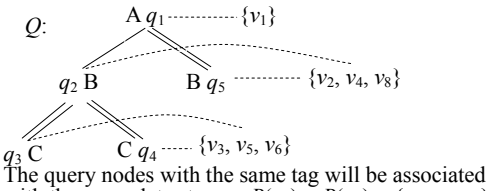


Fig. 5. Illustration for tree encoding.



The query nodes with the same tag will be associated with the same data stream: $B(q_2) = B(q_5) = \{v_2, v_4, v_8\}$

Fig. 6. Illustration for $B(q_i)$'s.

In Fig. 5, 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 $d_1 = d_2$, $l_1 \le l_2$, and $r_1 \ge r_2$, we say that α_2 is subsumed by α_1.

For convenience, a quadruple is considered to be subsumed by itself. If no confusion

is caused, we will used v and $\alpha(v)$ interchangeably. We also use $T[v]$ to represent a subtree rooted at v in T.

3. ORDERED TREE PATTERN MATCHING

In this section, we discuss our strategy for the ordered tree pattern matching. First, we discuss an algorithm for *subtree reconstruction* according to a given set of data streams in section 3.1. Then, in section 3.2, we extend this algorithm to evaluate ordered tree pattern queries by using a new tree labeling technique for queries, which enables us to handle left-to-right relationships in an efficient way. The index-based version of the algorithm will be discussed in section 4.

3.1 Tree Reconstruction

As with *TwigStack* [4], 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.*, $label(v) = label(q)$). All the quadruples in a data stream are sorted by their (DocID, LeftPos) values. For example, in Fig. 6, 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. 5, sorted according to their (DocID, LeftPos) values. For simplicity, we use the node names in a list, instead of the node's quadruples.

Note that iterating through the stream nodes in the sorted order of their LeftPos values corresponds to access of the document nodes in preorder (top-down). However, visiting the document nodes in preorder does not support an efficient check of tree matching since to know whether a query subtree rooted at some node q matches a document subtree rooted at a node v, we need first to know whether any subtree of q matches a subtree of v. This character hints that it is better to visit the documents nodes in *postorder* (*i.e.*, in the sorted order of their RightPos values). For this reason, we maintain a global stack ST to make a transformation of data streams using the following algorithm. In ST, each entry is a pair (q, v) with $q \in Q$ 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 $L(q_i)$; }
7. $ST.push(q_i, v)$;
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 each of them at the end of a $L(q_i)$'s (see lines 3 and 4); and push *v* into *ST* (see line 7). Then, the elements in each $L(q_i)$ must be 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. But for ease of explanation, we assume that all $L(q_i)$'s are completely generated in the following discussion. We also note that the data streams associated with different nodes in *Q* may be the same. (For example, in Fig. 6, we have $B(q_2) = B(q_5) = \{v_2, v_4, v_8\}$.) So we use $\boldsymbol{q}$ to represent the set of such query nodes and denote by $L(\boldsymbol{q})$ ($B(\boldsymbol{q})$) 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 $L(Q) = \{L(\boldsymbol{q}_1), \ldots, L(\boldsymbol{q}_l)\}$ (resp. $B(Q) = \{B(\boldsymbol{q}_1), \ldots, B(\boldsymbol{q}_l)\}$) to represent all the data streams with respect to *Q*, where each $\boldsymbol{q}_i$ ($i = 1, \ldots, l$) is a set of sorted query nodes that share a common data stream.

First, we discuss how to reconstruct a subtree structure from the data streams, based on the concept of *matching subtrees*, defined below.

Let *T* be a tree and *v* be a node in *T* with parent node *u*. Denote by *delete*(*T*, *v*) the tree obtained from *T* by removing node *v*. The children of *v* become 'descendant' children of *u*. (See Fig. 7 for illustration.)

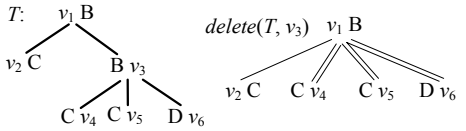


Fig. 7. The effect of removing v_3 from *T*.

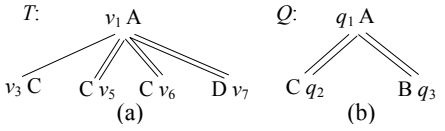


Fig. 8. Illustration for matching subtrees.

Definition 4 *matching subtrees*: A matching subtree *T'* of *T* with respect to a tree pattern *Q* is a tree obtained by a series of deleting operations to remove any node in *T*, which does not match any node in *Q*. ❑

For example, the tree shown in Fig. 8 (a) is a matching subtree of the document tree shown in Fig. 5 with respect to the query tree shown in Fig. 8 (b).

Given $L(Q)$, what we want is to construct a matching subtree from them to facilitate the checking of tree pattern matchings.

The algorithm given below handles the case when the streams contain nodes from a single XML document. When the streams contain nodes from multiple documents, the algorithm is easily extended to test equality of DocID before manipulating the nodes in the streams.

We will execute an iterative process to access the nodes in $L(Q)$ one by one:

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) ← v. Finally, set *left-sibling*(v) ← v'' and remove the links along the left- sibling chain.

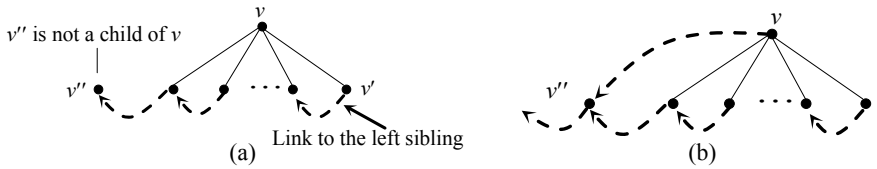


Fig. 9. Illustration for construction of matching subtrees.

Fig. 9 is a pictorial illustration of this process. In Fig. 9 (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. 9 (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.

Below is the formal description of the algorithm, which needs only $O(|D| \cdot |Q|)$ time. We elaborate this process since it can be extended to an efficient algorithm for evaluating ordered tree pattern queries.

Algorithm *matching-tree-construction*($L(Q)$)
Input: all data streams $L(Q)$.
Output: a matching subtree T'.
begin
1. **repeat until** each $L(\boldsymbol{q})$ in $L(Q)$ becomes empty
2. { identify $\boldsymbol{q}$ such that the first element v of $L(\boldsymbol{q})$ is of the minimal RightPos value;
3. remove v from $L(\boldsymbol{q})$; call *construction*(v, $\boldsymbol{q}$); }
end

Algorithm *construction*(v, $\boldsymbol{q}$)
begin
1. generate node v;
2. **if** v is not the first node created **then**
3. { **let** v' be the node generated just before v;
4. **if** v' is not a child (descendant) of v **then**
5. { *left-sibling*(v) ← v'; } (* generate a left-sibling link. *)
6. **else**
7. { v'' ← v'; w ← v'; (* v'' and w are two temporary variables. *)
8. **while** v'' is a child (descendant) of v **do**
9. { *parent*(v'') ← v; (* generate a parent link. Also, indicate whether v'' is a /-child or a //-child. *)
10. w ← v''; v'' ← *left-sibling*(v'');

```
11.         }
12.       left-sibling(v) ← v′′;
13.     }
14. }
end
```

In the above algorithm, for each chosen v from a $L(\boldsymbol{q})$, a node is created. At the same time, a left-sibling link of v is established, pointing to the node v' that is generated before v, if v' is not a child (descendant) of v (see line 5 in *construction*( )). Otherwise, we go into a **while**-loop to travel along the left-sibling chain starting from v' until we meet a node v'' which is not a child (descendant) of v. During the process, a parent link is generated for each node encountered except v'' (see lines 7-11). Finally, the left-sibling link of v is set to be v'' (see line 12).

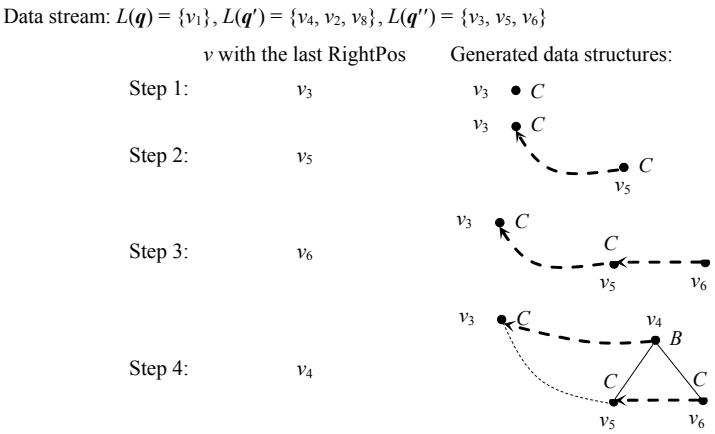


Fig. 10. Sample trace.

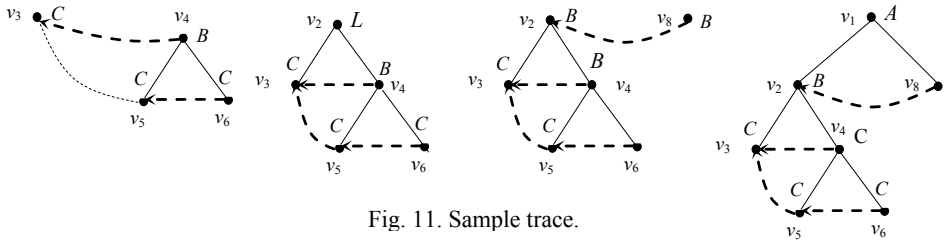


Fig. 11. Sample trace.

Example 1: Consider the tree pattern shown in Fig. 5 once again. After the data stream transformation, we will have three different data streams: $L(\boldsymbol{q}) = \{v_1\}$, $L(\boldsymbol{q}') = \{v_4, v_2, v_8\}$, $L(\boldsymbol{q}'') = \{v_3, v_5, v_6\}$, where $\boldsymbol{q} = \{q_1\}$, $\boldsymbol{q}' = \{q_2, q_5\}$, $\boldsymbol{q}'' = \{q_3, q_4\}$. Applying the above algorithm to the data streams, we generate a series of data structures as shown in Fig. 10.

In step 1 (see Fig. 10), v_3 is checked since it has the least RightPos; and a node for it is created. In step 2, we meet v_5. Since it is not a descendant of v_3, we establish a left-sibling link from v_5 to v_3. In step 3, we generate node v_6 and a left-sibling link from v_6 to v_5. In

step 4, we generate part of the matching tree, in which two edges from v_4 respectively to v_5 and v_6 are created. Special attention should be paid to step 4. In this step, not only two edges are constructed, but a left-sibling link from v_4 to v_3 is also created. It is this kind of left-sibling links that enable us to reconstruct a matching subtree in an efficient way.

The subsequent computation is shown in Fig. 11. ❑

Proposition 1 Let T be a document tree. Let Q be a tree pattern. Let $L(Q) = \{L(\boldsymbol{q}_1), \ldots, L(\boldsymbol{q}_l)\}$ be all the data streams with respect to Q and T, where each $\boldsymbol{q}_i$ $(1 \le i \le l)$ is a subset of sorted query nodes of Q, which share the same data stream. Algorithm *matching-tree-construction*($L(Q)$) generates the matching subtree T' of T with respect to Q correctly.

Proof: Denote $L = |L(\boldsymbol{q}_1)| + \ldots + |L(\boldsymbol{q}_l)|$. We prove the proposition by induction on L.

Basis When $L = 1$, the proposition trivially holds.

Induction Hypothesis Assume that when $L = k$, the proposition holds.

Induction Step We consider the case when $L = k + 1$. Assume that all the quadruples in $L(Q)$ are $\{u_1, \ldots, u_k, u_{k+1}\}$ with RightPos(u_1) < RightPos(u_2) < … < RightPos(u_k) < RightPos(u_{k+1}). The algorithm will first generate a tree structure T_k for $\{u_1, \ldots, u_k\}$. In terms of the induction hypothesis, T_k is correctly created. It can be a tree or a forest. If it is a forest, all the roots of the subtrees in T_k are connected through left-sibling links. When we meet v_{k+1}, two cases need to be considered:

Case 1: v_{k+1} is an ancestor of v_k,
Case 2: v_{k+1} is to the right of v_k.

In case 1, the algorithm will generate an edge (v_{k+1}, v_k), and then travels along a left-sibling chain starting from v_k until we meet a node v which is not a descendant of v_{k+1}. For each node v' encountered, except v, an edge (v_{k+1}, v') will be generated. Therefore, T_{k+1} is correctly constructed. In case 2, the algorithm will generate a left-sibling link from v_{k+1} to v_k. It is obviously correct since in this case v_{k+1} cannot be an ancestor of any other node. This completes the proof. ❑

The time complexity of this process is easy to analyze. First, we notice that each quadruple in all the data streams is accessed only once. Secondly, for each node in T', all its child nodes will be visited along a left-sibling chain for a second time. So we get the total time

$$O(|D| \cdot |Q|) + \sum_i d_i = O(|D| \cdot |Q|) + O(|T'|) = O(|D| \cdot |Q|),$$

where d_i represents the outdegree of node v_i in T'.

During the process, for each encountered quadruple, a node v will be generated. Associated with this node have we at most two links (a left-sibling link and a parent link). So the used extra space is bounded by $O(|T'|)$.

3.2 Algorithm for Ordered Tree Matching

In fact, the algorithm discussed in section 3.1 hints an efficient way for processing ordered tree pattern queries.

We first observe that during the reconstruction of a matching subtree T', we can also associate each node v in T' with a *query node stream* S_v. That is, each time we choose a v with the least RightPos value from a data stream $L(\boldsymbol{q})$, we will insert all the query nodes in $\boldsymbol{q}$ into S_v. For example, in the first step shown in Fig. 10, the query node stream for v_3 can be determined as shown in Fig. 12 (a).

In this way, we can create a matching subtree as shown in Fig. 12 (b), in which each node in T' is associated with a query node stream. 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.

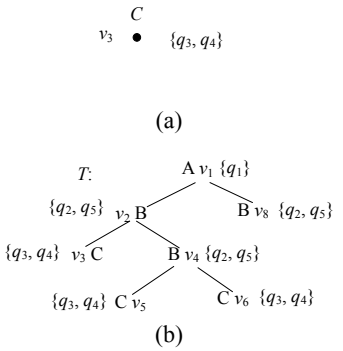


Fig. 12. Illustration for generating *QS*'s.

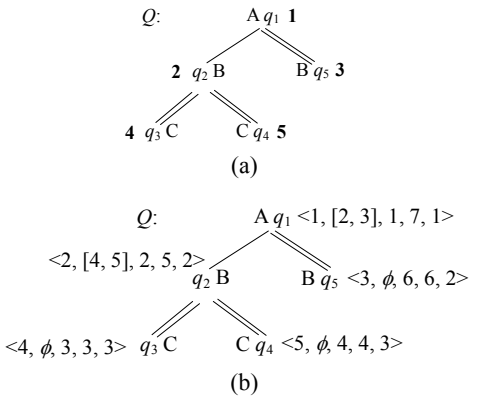


Fig. 13. Illustration for labeling *Q*.

For this purpose, we will first search Q in the breadth-first fashion, generating a number (called a *breadth-first number*) for each node q in Q, denoted as $bf(q)$, which represents the left-to-right order of siblings in a simple way (see Fig. 13 (a) for illustration). Then, we use $interval(q)$ to represent an interval covering all the breadth-first numbers of q's children. For example, for Q shown in Fig. 13 (a), we have $interval(q_1) = [2, 3]$ and $interval(q_2) = [4, 5]$. In the following, we will use q and $bf(q)$ interchangeably.

Next, we associate each q with a tuple $g(q) = <bf(q), interval(q), \text{LeftPos}(q), \text{RightPos}(q), \text{LevelNum}(q)>$, as shown in Fig. 13 (b). We say, a q is subsumed by a pair (L, R) if $L \le \text{LeftPos}(q)$ and $R \ge \text{RightPos}(q)$. When checking the tree embedding of Q in T', we will associate each generated node v in T' with a linked list A_v to record what subtrees in Q can be embedded in $T'[v]$. Each entry in A_v is a quadruple $e = (q, interval, L, R)$, where q is a node in Q, $interval = [a, b] \subseteq interval(q)$ (for some $a \le b$), $L = \text{LeftPos}(a)$ and $R = \text{RightPos}(b)$. Here, we use a and b to refer to the nodes with the breadth-first numbers a and b, respectively. Therefore, such a quadruple represents a set of subtrees (in $Q[q]$) rooted respectively at a, $a + 1$, …, b (*i.e.*, a set of subtrees respectively rooted at a set of consecutive breadth-first numbers). See Fig. 14 for illustration.

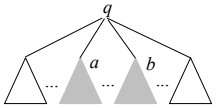


Fig. 14. Subtrees represented by *bf*-numbers.

In addition, the following two conditions are satisfied:

(1) For any two entries e_1 and e_2 in A_v, $e_1.q$ is not subsumed by ($e_2.L$, $e_2.R$), nor is $e_2.q$ subsumed by ($e_1.L$, $e_1.R$). In addition, if $e_1.q = e_2.q$, $e_1.interval \not\subset e_2.interval$ and $e_2.interval \not\subset e_1.interval$.
(2) For any two entries e_1 and e_2 in A_v with $e_1.interval = [a, b]$ and $e_2.interval = [a', b']$, if e_1 appears before e_2, then RightPost($e_1.q$) < RightPost($e_2.q$) or RightPost($e_1.q$) = RightPost($e_2.q$) but $a < a'$.

Condition (1) is used to avoid redundancy due to the following lemma.

Lemma 1 Let q be a node in Q. Let $[a, b]$ be an interval. If q is subsumed by (LeftPos(a), RightPos(b)), then there exists an integer $0 \le i \le b - a$ such that $bf(q)$ is equal to $a + i$ or q is an descendant of $a + i$.

Proof: The proof is trivial. ❑

Then, by imposing condition (1), A_v keeps only quadruples which represent pairwise non-covered subtrees.

Condition (2) is met if the nodes in Q are checked along their increasing RightPos values. It is because in such an order the parents of the checked nodes must be non-decreasingly sorted by their RightPos values. Since we explore Q bottom-up (note that each ***q*** is sorted increasingly by RightPos values), condition (2) is always satisfied.

See Fig. 15 for a better understanding.

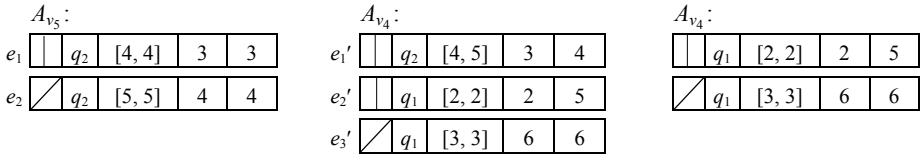


Fig. 15. Illustration for linked lists associated with nodes in T.

Fig. 15 (a) shows the linked list created for v_5 in T' shown in Fig. 12 (b) when it is generated and checked against q_3 and q_4 in Q shown in Fig. 13 (b). Since both q_3 and q_4 are leaf nodes, $T'[v_5]$ is able to embed either $Q[q_3]$ or $Q[q_4]$ and so we have two entries e_1 and e_2 in A_{v_5} Note that $bf(q_3) = 4$ and $bf(q_3) = 5$. So we set their *intervals* to [4, 4] and [5, 5], respectively. In addition, each of them is a child of q_2. Thus, we have $e_1.q = e_2.q = q_2$. In Fig. 15 (b), we show the linked list for v_4. It contains three entries e_1', e_2' and e_3'. Special attention should be paid to e_1'. Its *interval* is [4, 5], showing that $T'[v_4]$ is able to

embed both $Q[q_3]$ and $Q[q_4]$. In this case, $e_1'.L$ is set to 3 and $e_1'.R$ to 4. However, since $e_1'.q = q_2$ is subsumed by $(e_2'.L, e_2'.R) = (2, 5)$, the entry will be removed, and the linked list is reduced to a data structure shown in Fig. 15 (c).

With the linked lists associated with the nodes in T', the embedding of a subtree $Q[q]$ in $T'[v]$ can be checked very efficiently. First, we define a simple operation over two intervals $[a, b]$ and $[a', b']$, which share the same parent:

$$[a,b]\Delta[a',b'] = \begin{cases} [a,b'], & \text{if } a \le a' \le b+1, b < b'; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

For example, in A_{v_5}, we have an entry $(q_2, [4, 4], 3, 3)$. In A_{v_6} (which is exactly the same as A_{v_5}), we have an entry $(q_2, [5, 5], 4, 4)$. We can merge these two entries to form another entry $(q_2, [4, 5], 3, 4)$, which can be used to facilitate checking whether $T'[v_4]$ embeds $Q[q_2]$.

The general process to merge two linked list is described below.

1. Let A_1 and A_2 be two linked list associated with the first two child nodes of a node v in T', which is being checked against q with $label(v) = label(q)$.
2. Scan both A_1 and A_2 from the beginning to the end. Let e_1 (from A_1) and e_2 (from A_2) be the entries encountered. We will perform the following checkings.
   – If RightPos($e_2.q$) > RightPos($e_1.q$), $e_1 \leftarrow next(e_1)$.
   – If RightPos($e_2.q$) < RightPos($e_1.q$), then $e_2' \leftarrow e_2$; insert e_2' into A_1 just before e_1; $e_2 \leftarrow next(e_2)$.
   – If RightPos($e_2.q$) = RightPos($e_1.q$), then we will compare the intervals in e_1 and e_2. Let $e_1.interval = [a, b]$. Let $e_2.interval = [a', b']$.
     If $a' > b + 1$, then $e_1 \leftarrow next(e_1)$.
     If $a \le a' \le b + 1$ and $b < b'$, then replace $e_1.interval$ with $[a, b] \Delta [a', b']$ in A_1; e_1.RightPost $\leftarrow$ RightPos(b'); $e_1 \leftarrow next(e_1)$; $e_2 \leftarrow next(e_2)$.
     If $[a', b'] \subseteq [a, b]$, then $e_2 \leftarrow next(e_2)$.
     If $a' < a$, then $e_2' \leftarrow e_2$; insert e_2' into A_1 just before e_1; $e_2 \leftarrow next(e_2)$.
3. If A_1 is exhausted, all the remaining entries in A_2 will be appended to the end of A_1.

The result of this process is stored in A_1, denoted as $merge(A_1, A_2)$. We also define

$$merge(A_1, \ldots, A_k) = merge(merge(A_1, \ldots, A_{k-1}), A_k),$$

where $A_1, \ldots, A_k$ are the linked lists associated with v's child nodes: $v_1, \ldots, v_k$, respectively. If in $merge(A_1, \ldots, A_k)$ there exists an e such that $e.interval = interval(q)$, $T'[v]$ embeds $Q[q]$.

For the merging operation described above, we require that the entries in a linked list are sorted. That is, all the entries e are in the order of increasing RightPos($e.q$) values; and for those entries with the same RightPos($e.q$) value their intervals are 'from-left-to-right' ordered. Such an order is obtained by searching Q bottom-up (or say, in the order of increasing RightPos values) when checking a node v in T' against the nodes in Q. Thus, no extra effort is needed to get a sorted linked list. Moreover, if the input linked lists are sorted, the output linked lists must also be sorted.

In terms of the above discussion, we give our algorithm for evaluating ordered tree pattern queries. As with the algorithm *matching-tree-construction*( ), we will generate *left-sibling* links to facilitate the computation. However, in the following description, we focus on the checking of tree embedding and that part of technical details is omitted.

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
3. RightPos value; remove v from $L(\boldsymbol{q})$; call *embedding*(v, $\boldsymbol{q}$);
4. }
end

Algorithm *embedding*(v, $\boldsymbol{q}$)
begin
1. **generate node** v; $A_v \leftarrow \phi$;
2. let v_1, …, v_k be the children of v.
3. $\boldsymbol{A} \leftarrow merge(A_{v_1}, \ldots, A_{v_k})$;
4. **for** each $q \in \boldsymbol{q}$ **do** { (* nodes in $\boldsymbol{q}$ are sorted. *)
5. **if** q is a leaf **then** $\{S_v \leftarrow S_v \cup \{q\};\}$
6. **else** (* q is an internal node. *)
7. { **if** there exists e in $\boldsymbol{A}$ such that $e.interval = interval(q)$
8. **then** $S_v \leftarrow S_v \cup \{q\}$; }
9. }
10. **for** each $q \in S_v$ **do** {
11. *append* (q's parent, [$bf(q)$, $bf(q)$)], q.LeftPos, q.RightPos) to the end of A_v; }
12. $A_v \leftarrow merge(A_v, A)$; Scan A_v to remove subsumed entries;
13. remove all A_{v_k}'s;
14. }
15. }
end

In the above algorithm, the nodes in T' is created one by one as done in Algorithm *matching-tree-construction*( ). But for each node v generated for an element from a $L(\boldsymbol{q})$, we will first merge all the linked lists of their children and store the output in a temporary variable A (see line 3 in Algorithm *embedding*( )). Then, for each $q \in \boldsymbol{q}$, we will check whether there exists an entry e such that $e.interval = interval(q)$ (see lines 7 and 8). If it is the case, we will construct an entry for q and append it to the end of the linked list A_v (see lines 10 and 11). The final linked list for v is established by executing line 12. Afterwards, all the A_{v_k}'s (for v's children) will be removed since they will not be used any more (see line 13).

In the same way, we will generate A_{v_5} and A_{v_6} as shown in Figs. 16 (b) and (c), respectively. When we create v_4 (taken from $L(\boldsymbol{q}') = \{v_4, v_2, v_8\}$ with $\boldsymbol{q}' = \{q_2, q_5\}$), we will first merge A_{v_5} and A_{v_1} to generate a linked list as shown in Fig. 16 (d). Since the interval in

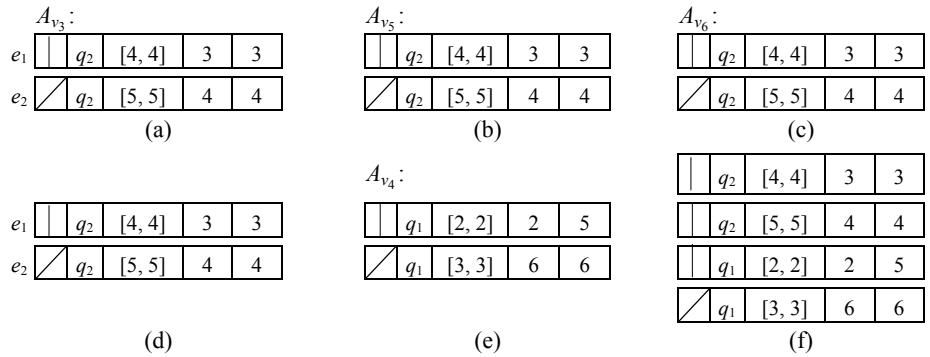


Fig. 16. Illustration for linked lists associated with nodes in T.

the first entry in it is equal to $interval(q_2)$, we know that $T'[v_4]$ embeds $Q[q_2]$. As described above, the final linked list for v_4 will be a list as shown in Fig. 16 (e). When we generate v_2 (taken from $L(\boldsymbol{q}') = \{v_4, v_2, v_8\}$) we will first merge A_{v_3} and A_{v_4}, and get a linked list as shown in Fig. 17 (f). Since in this linked list we do not have an entry e with $e.interval = interval(q_2)$, we will not construct an entry for q_2. But in the linked list shown in Fig. 16 (f), we already have an entry for q_2 (remember that $bf(q_2) = 2$), showing that there must exist a subtree rooted at one of v_2's descendant, which embeds $Q[q_2]$. In this way, any redundant work can be avoided. Continuing this process, we will find out that T' embeds Q.

The above algorithm can be used only for the case that Q contains no /-edges. In the presence of both //-edges and /-edges, it should be slightly modified as below.

Let $q_1, \ldots, q_k$ be the children of q; and among them $q_{i_1}, \ldots, q_{i_j}$ be all the /-children. Let v_l be a child of v and $e = <bf(q), [a, b], \text{LeftPos}(a), \text{RightPos}(b)>$ be an entry in A_{v_l}. If v_l is a /-child of v, e will not need be changed. Otherwise, we will find the first q_{i_c} such that $bf(q_{i_c}) = x \geq a$ and the last q_{i_d} such that $bf(q_{i_d}) = y \leq b$; and replace e in A_{v_l} with the following entries before it takes part in the merging operation:

$<bf(q), [a, bf(q_{i_c}) - 1], \text{LeftPos}(a), \text{RightPos}(bf(q_{i_c}) - 1)>$,
$<bf(q), [bf(q_{i_c}) + 1, bf(q_{i_{c+1}}) - 1], \text{LeftPos}(bf(q_{i_c}) + 1), \text{RightPos}(bf(q_{i_{c+1}}) - 1)>$,
…
$<bf(q), [bf(q_{i_d}) + 1, b], \text{LeftPos}(bf(q_{i_d}) + 1), \text{RightPos}(bf(b))>$.

It is because in the case that v_l is a //-child, we should remove the points $bf(q_{i_c}), \ldots, bf(q_{i_d})$ from $[a, b]$, as if $T'[v_l]$ is not able to cover $Q[q_{i_c}], \ldots, Q[q_{i_d}]$.

Concerning the correctness of the algorithm, we have the following proposition.

Proposition 2 Algorithm *tree-embedding*( ) computes the entries 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 sorted.

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 $v_1, \ldots, v_k$ be the children of v. Then, for each v_i ($i = 1, \ldots, k$), we have $h(v_i) \leq l$. In terms of the induction hypothesis, each A_{v_i} is correctly constructed and sorted. Then, the output of $merge(A_{v_1}, \ldots, A_{v_k})$ is sorted. If there exists an e such that $e.interval = interval(q)$ for some q with $label(v) = label(q)$, an entry for q will be constructed and appended to the end of A_v. Again, since the nodes in Q are checked in the order of increasing RightPos values, A_v must be sorted. So $merge(A_v, merge(A_{v_1}, \ldots, A_{v_k}))$ is correctly constructed and sorted. ❑

Now we analyze the time complexity of the algorithm. First, we see that for each node v in T', d_v merging operations will be conducted, where d_v is the outdegree of v. The cost of a merging operation is bounded by $O(leaf_Q)$ since the length of each linked list A_v associated with a node v in T' is bounded by $O(leaf_Q)$ according to the following analysis. Consider two nodes q_1 and q_2 on a path in Q, if both $Q[q_1]$ and $Q[q_2]$ can be embedded in $T'[v]$, A_v keeps only one entry for them. If q_1 is an ancestor of q_2, then A_v contains only the entry for q_1 since embedding of $Q[q_1]$ in $T'[v]$ implies the embedding of $Q[q_2]$ in $T'[v]$. Otherwise, A_v keeps only the entry for q_2. Obviously, Q can be divided into exactly $leaf_Q$ root- to-leaf paths. Furthermore, the merge of two linked lists A_1 and A_2 takes only $O(\max\{|A_1|, |A_2|\})$ time since both A_1 and A_2 are sorted lists according to the proof of Proposition 2. (It works in a way similar to the *sort-merge join*.) Therefore, the cost for generating all the linked lists is bounded by

$$O(\sum_{v \in T'} d_v \cdot leaf_Q) = O(|T'| \cdot leaf_Q).$$

In addition, for each node v taken from a $L(\boldsymbol{q})$, each q in $\boldsymbol{q}$ will be checked (see line 4 in Algorithm *embedding*( ).) This part of checking can be slightly improved as follows. Let $L(\boldsymbol{q}) = \{q_1, \ldots, q_k\}$. Each q_j ($j = 1, \ldots, k$) is associated with an interval $[a_j, b_j]$. Since q_j's are sorted by RightPos values, we can check A (= $merge(A_{v_1}, \ldots, A_{v_k})$) against $\boldsymbol{q}$ in one scanning to find, for each q_j, whether there is an interval in A, which is equal to $[a_j, b_j]$. This process needs only $O(|A| + |\boldsymbol{q}|)$ time. So the total cost of this task is bounded by $O(|T'| \cdot leaf_Q) + O(|D| \cdot |Q|)$.

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

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

The space overhead of Algorithm *tree-embedding*( ) is in the order of $O(leaf_{T'} \cdot leaf_Q)$. It is because at any time point during the execution, at most $leaf_{T'}$ nodes in T' are associated with a linked list (see line 15 in Algorithm *tree-embedding*( )).

In the above discussion, the main algorithm has been described in detail. However, two issues yet remain to be addressed. That is, the treatment of the wildcards (*) as well as the output node in Q should be made clear.

In fact, using XB-trees, * is handled in the same way as non-wildcard nodes. As we will see in the next section, 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 *drilldown* and *advance* operators [4], any entry in an XB-tree is accessed only

once.

As for the output node of Q, we should notice that the set S_v generated for each node v in T' (such that for each $q \in S_v$ $T'[v]$ embeds $Q[q]$) does not serve as the answer to Q. But in the algorithm we can maintain two additional data structures: G_r and G_o. G_r contains all those document nodes v such that $Q[r]$ can be embedded in $T'[v]$, where r is the root of Q; and G_o contains all the document nodes u such that $Q[o]$ can be embedded in $T'[u]$, where o is the output node of Q. We can sort G_r and G_o such that all nodes are increasingly sorted by the RightPos values. Then, based on these two data streams, we can create another subtree T'' of T' (in a way similar to the generation of a matching subtree), which 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 not a /-child of r. Otherwise an o-node has to be a /-child of some r-node to be an answer.

4. INDEX-BASED ALGORITHM

In the algorithm discussed in the previous section, we need to access all the nodes in a $B(\boldsymbol{q})$ for each $\boldsymbol{q} \subset Q$. To improve this, we incorporate the XB-tree [4] into our algorithm. However, since XB-tree is initially established for checking the unordered tree matching, some changes have to be made for the purpose of ordered tree matching.

First, we notice that the actual input of our algorithm is $B(\boldsymbol{q})$'s (in which the nodes are sorted by LeftPos values, see section 3.1). Thus, we can construct an XP-tree for each $B(\boldsymbol{q})$ in the same way as *TwigStackXB* [4]. An XB-tree is a variant of B^+-tree over a quadruple sequence. In such an index structure, each entry in a page is a pair a = (LeftPos, 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. 17 (a), for which we may generate an XB-tree as illustrated in Fig. 17 (b).

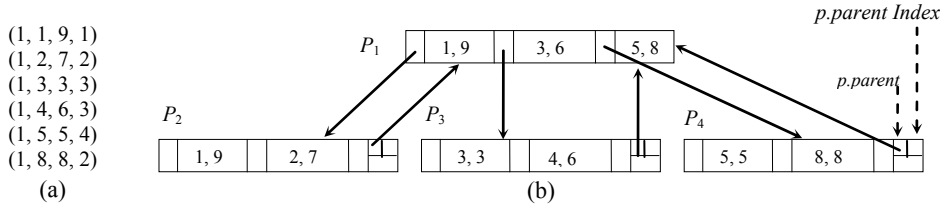


Fig. 17. A quadruple sequence and XB-tree over it.

In each page P of an XB-tree, the bounding segments may partially overlap, but their LeftPos positions are in increasing order. Let a_1 and a_2 be two consecutive entries. Then the leftPos value of any entry in the subtree rooted at a_1 is smaller than the leftPos value of a_2. For instance, in the XB-tree shown in Fig. 17 (b), the first entry of P_1: [1, 9] overlaps the second entry: [3, 6]. But the leftPos value of any entry in the subtree rooted at [1, 9] is smaller than 3. 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 ith pointer in $P.parent$ points to P. For instance, in the XB-tree shown in Fig. 17 (b), $P_3.parentIndex = 2$ since the second pointer in P_1 (the parent of P_3) points to P_3.

Consider an XB-tree constructed for $B(\boldsymbol{q})$ with $\boldsymbol{q} = \{q_1, \ldots, q_k\}$. For each $q_j \in \boldsymbol{q}$ ($j = 1, \ldots, k$), we maintain a pair (P, i), denoted β_{q_j}, to indicate that the ith entry in the page P is currently accessed for q_j. Thus, each β_{q_j} ($j = 1, \ldots, k$) corresponds to a different searching of the same XB-tree as if we have a separate copy of that XB-tree over $B(q_j)$.

In [4], two operations are defined to navigate an XB-tree, which change the value of β_q.

1. $advance(\beta_q)$ (going up from a page to its parent): If $\beta_q = (P, i)$ does not point to the last entry of P, $i \leftarrow i + 1$. Otherwise, $\beta_q \leftarrow (P.parent, P.parentIndex + 1)$.
2. $drilldown(\beta_q)$ (going down from a page to one of its children): If $\beta_q = (P, i)$ and P is not a leaf page, $\beta_q \leftarrow (P', 1)$, where P' is the ith child page of P.

Initially, for each q, β_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 β_q to ϕ, showing that the XB-tree over $B(q)$ is exhausted.) As with *TwigStackXB*, in each step we will determine a $q \in Q$ and take an entry from the corresponding XB-tree. But an entry taken from an XB-tree can be an entry in a non-leaf node, which does not correspond to an element in documents. So we use the following function to check it:

$isPlainValue(\beta_q)$: returns *true* if β_q is pointing to a leaf node in the corresponding XB-tree.

If β_q points to an entry in a non-leaf node, we need to navigate the XB-tree. (The pruning happens when we advance β_q. In this case, the subtree pointed to by the current entry is skipped over. See below). Otherwise, it points to an entry in a leaf node. The entry (representing a document node) is then pushed into *ST* as done in Algorithm *stream- transformation*( ). Besides, each $q \in Q$ is associated with an extra linked list, denoted $link_q$, such that each entry in it contains a pointer to a node v stored in *ST* with $label(v) = label(q)$. We append entries to the end of a $link_q$ one by one as the document nodes are inserted into *ST*, as illustrated in Fig. 18 (a). When a node is popped out from *ST*, the corresponding entry is removed from the corresponding $link_q$.

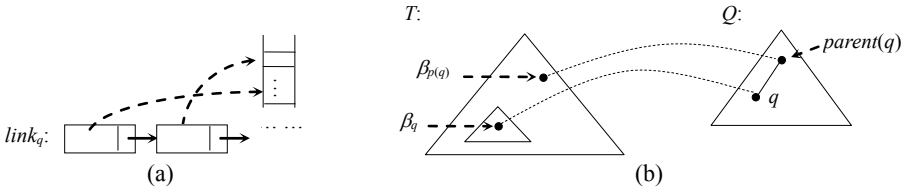


Fig. 18. Illustration for $advance(\beta_q)$.

In the index-based version of our main algorithm, each time to determine a q to check, we call a function *getNext*( ), which returns a query node. In addition, the following three functions are also used:

end(Q): if for each leaf node q of Q $\beta_q = \phi$ (*i.e.*, $L(q)$ is exhausted), then returns *true*; otherwise, *false*.
currL(β_q): returns the LeftPos of the entry pointed to by β_q.
currR(β_q): returns the RightPos of the entry pointed to by β_q.
isRoot(q): if q is the root, return *true*; otherwise, *false*.
isLeaf(q): returns *true* if q is a leaf of Q; otherwise, *false*.
$p(q)$: returns the parent of q.
$l(q)$: returns $\delta(q) - 1$, where $\delta(q)$ is the postorder number of the left-most leaf node in $Q[q]$. Note that $\delta(q) - 1$ is the largest postorder number to the left of q.

Algorithm *XB-tree-matching*(*XB-trees over* $B(q_i)$'s)
Input: *XB-trees*(q_i)'s.
Output: S_v's, which show the tree embedding.
begin
1. **while** ($\neg$*end*(Q)) **do**
2. { $q \leftarrow$ *getNext*(*root-of-Q*);
3. **if** (*isPlainValue*(β_q) **then**
4. { let v be the node pointed to by β_q;
5. **while** *ST* is not empty and *ST.top* is not v's ancestor **do**
6. { $x \leftarrow$ *ST.pop*(); Let $x = (q', u)$;
7. call *embedding*(u, q'); }
8. *ST.push*(q, v); *advance*(β_q);
9. }
10. **else if** ($\neg$*isRoot*(q) $\wedge$ $link_{p(q)} = \phi \wedge$ *currR*(β_q) < *currL*($\beta_{p(q)}$))
11. $\vee$ $l(q) \neq 0 \wedge$ *currL*(β_q) $\leq$ *currR*($\beta_{l(q)}$))
12. **then** *advance*(β_q) (* not part of a solution *)
13. **else** *drilldown*(β_q); (* may find a solution *)
14. }
end

In the above algorithm, we distinguish between two cases. If β_q is an entry in a leaf node in the corresponding XB-tree, we will insert it into *ST* (see lines 3 and 8). Each time an element is popped out of *ST*, Algorithm *embedding*( ) will be invoked to check a subtree embedding (see lines 6 and 7).

If β_q is an entry in a non-leaf node, lines 10-13 will be carried out. If *currR*(β_q) < *currL*($\beta_{p(q)}$) (see line 10), we have a situation as illustrated in Fig. 18 (b): for the current node v pointed to by β_q (which matches q, we cannot find an ancestor of v, which matches q's parent. In this case, we need to advance β_q (see line 12.) If *currL*(β_q) $\leq$ *currR*($\beta_{l(q)}$) (see line 11), the left-to-right ordering is violated and we also need to advance β_q. In this way, a lot of useless access of document nodes can be saved. If it is not the case, we will drill down the corresponding XB-tree (see line 12) since a solution may be found.

In the following, we discuss *getNext*( ) in great detail to see how a $q \in Q$ is determined in each step to take an entry from an XB-tree.

According to [4], each time we determine a q ($\in Q$), for which an entry from $B(q)$ is taken, the following three conditions are checked:

(1) For q, there exists an entry v_q in $B(q)$ such that it has a descendant v_{q_i} in each of the streams $B(q_i)$ (where q_i is a child of q).
(2) Each v_{q_i} recursively satisfies (1).
(3) LeftPos(v_q) is minimum.

However, since by the ordered tree matching the order of siblings is significant, we can impose a fourth condition to skip more entries in an XB-tree. For this purpose, for each $q \in Q$, we establish *left-sibling*(q) in the same way as described in section 3.1. Then, we have

(4) If $l(q) \neq \phi$, v_q in $B(q)$ is to the right of the entry pointed to by $\beta_{l(q)}$.

According to the above conditions, we design our own *getNext*( ), which is a modification of the corresponding algorithm discussed in [4]. It is in essence a bottom-up search of Q, but by a recursive process starting from the root of Q. That is, a q is accessed after all its children have been visited by recursive calls. The algorithm *getNext*(q) works as follows:

(1) Once a q, for which β_q is pointing to an entry in a non-leaf node in the corresponding XB-tree, is encountered, it is immediately returned to *XB-tree-matching*( ) (see line 4 in *getNext*( )), by which *advance*(β_q) or *drilldown*(β_q) will be performed (see line 10 in *XB-tree-matching*( )).
(2) If for a q we cannot find, by performing *advance*(β_q), a v such that for all its children $q_i\beta_{q_i}$ is pointing to an entry which is a descendant of v, and condition (4) is satisfied, then $q_{\min}$ is returned, where q_{min} represents the return value of a recursive call *getNext*(a child of q) such that $\beta_{q_{\min}}$ points to an entry with the least LeftPos value. Otherwise, q returns as the output of *getNext*(q).

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. { $r_i \leftarrow$ *getNext*(q_i);
4. **if** ($r_i \neq q_i \vee \neg$*isPlainValue*(β_{r_j})) **then** return r_i;}
5. $q_{\min} \leftarrow q''$ such that $currL(\beta_{q''}) = \min_i\{currL(\beta_{r_j})\}$;
6. $q_{\max} \leftarrow q'''$ such that $currL((\beta_{q'''}) = \max_i\{currL(\beta_{r_j})\}$;
7. **while** ($currR(\beta_q) < currL(\beta_{q_{\min}}) \vee \ (l(q) \neq 0 \wedge currL(\beta_q) \leq currR(\beta_{l(q)}))$ **do** *advance*(β_q);
8. **if** $currL(\beta_q) < currL(\beta_{r_{\max}})$ **then** return q;
9. **else** return $q_{\min}$; }
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 (1)-(4). It is a recursive process to search Q bottom up (see line 3). If for one of q's children $q_i\beta_{q_i}$ is pointing to an entry in a non-leaf node in the corresponding XB-tree, it is returned (see line 4). Especially, it goes back through all the recursive calls and returns as the output of the whole process. It occurs due to the condition '$r_i \neq q_i$' in line 4. Lines 5-9

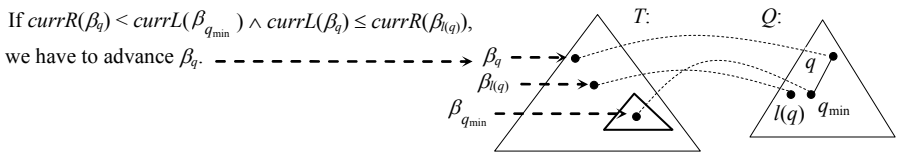


Fig. 19. Illustration for *advance*(β_q).

are used to check whether conditions (1) and (4) are satisfied. (See Fig. 19 for illustration of line 7.) In addition, the recursive call performed in line 3 shows that condition (2) is met. Since each XB-tree is traversed top-down and the entries in each node are scanned from left to right, condition (3) must always be satisfied.

The main difference of the modified *getNext*(q) from the corresponding algorithm discussed in [4] is in the checking condition: $l(q) \neq 0 \wedge currL(\beta_q) \le currR(\beta_{l(q)})$ in line 7, which enable us to skip over some subtrees in an XB-tree by checking the left-to-right ordering.

5. 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

In the experiments, we have tested four methods:

– *TwigStack* (*TS* for short) [4],
– *Twig*^{2}*Stack* (*T*^{2}*S* for short) [12],
– *PRIX* [30],
– *XB-tree-embedding* (discussed in this paper, *TE* for short).

The theoretical computational complexities of these methods are summarized in Table 1.

The index for *PRIX* is a *trie* structure over all the labeled Prüfer sequences, implemented as a B$^+$-tree [30]. The indexes for all the other three methods are XB-trees [4].

Table 1. Time and space complexities.

methods	query time	runtime space usage
TwigStack	$O(\|D\|^{\|Q\|})$	$O(\|D\| \cdot \|Q\|)$
Twig^2*Stack*	$O(\|D\| \cdot \|Q\|^2 + \|subTwigResults\|)$	$O(\|D\| \cdot \|Q\|)$
PRIX	$O(\|D\|^{\|Q\|})$	$O(\|D\| + \|Q\|)$
TE	$O(\|D\| \cdot \|Q\| + \|T'\| \cdot leaf_Q)$	$O(leaf_T \cdot \|Q\|)$

Table 2. Data sets for experimental evaluation.

	TreeBank	DBLP	XMark				
			1	2	3	4	5
Data size (MB)	82	127	113	228	340	454	568
Nodes (million)	2.43	3.33	1.72	3.33	5.1	6.7	8.33
Max/average depth	36/7.9	6/2.9	12/6.2				

Data

The data sets used for the tests are TreeBank data set [36], DBLP data set [36] and a synthetic XMARK data set [41]. 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 scaling factors of 1 to 5) is a well-known benchmark data set, which is used for scalability analysis. The important parameters of these data sets are summarized in Table 2.

For all the experiments, the buffer pool size was fixed at 2000 pages. The page size of 8KB was used. For each data set, all the tag names are stored in a single list and then each tag name is represented by its order number in that list during the evaluation of queries. In our implementation, each DocId occupies 4 bytes while a number in a Prüfer sequence, a LeftPos or a RightPos occupies 2 bytes. A levelNum value takes only 1 byte. In addition, in DBLP, each proceedings, journal volume and a book is considered to be a individual document, and therefore assigned a DocID.

Test results

In the experiments, we tested altogether 21 queries shown in Tables 3-5.

Table 3. queries for TreeBank data set.

query	XPath expression
*Q*1	//VP[DT]//PRP_DOLLAR
*Q*2	//S/VP/PP[IN]/NP
*Q*3	//S/VP//PP[NP/VB]/IN
*Q*4	//VP[.//PP/IN]//NP/*//JJ
*Q*5	//S[CC][.//PP]//NP[VBZ][IN]//JJ
*Q*6	//S[*/PRP]/VP[VBD]
*Q*7	//S[.//NNP]/VP[.//NP[.//NNP]]

Table 4. Queries for DBLP.

query	XPath expression
*Q*8	//article/authot="C.J. Codd"
*Q*9	//inproceedings[author="Jim Gray"][year="1990"]
*Q*10	//inproceedings[key][author="Jim Gray"][year="1990"]
*Q*11	//inproceeding[author][title][.//pages][.//url]
*Q*12	//articles[author][title][.//volume][.//pages][.//url]/*
*Q*13	//section[.//page]//year
*Q*14	//incollection[.//url]//booktitle

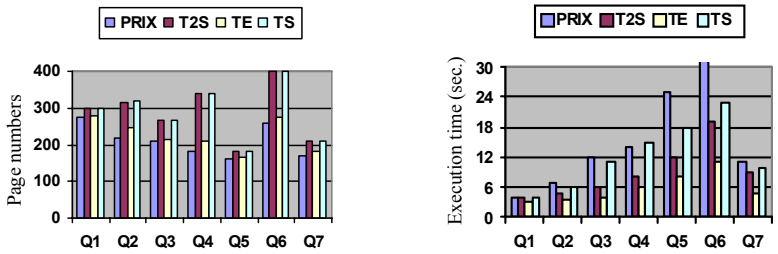
Table 5. Queries for XMark.

query	XPath expression
*Q*15	/site//open_auction[.//seller/person]/
*Q*16	/site//open_auction[.//seller/person][.//bidder]/
*Q*17	/site//open_auction[.//seller/person][.//bidder/increase]/
*Q*18	/site//open_auction[.//seller/person][.//bidder[increase][.//initial]]/
*Q*19	/site//open_auction[.//seller/person][.//bidder/increase][.//initial]/*/description/
*Q*20	//item[description]//mail
*Q*21	//people//*[homepage]/name

To avoid the frequent use of the axes like *following-sibling* in the tables, we assume that the order between the siblings in a tree query follows the left-to-right order in the corresponding XPath expression. For example, //inproceedings[key][author] indicates that *key* is followed by *author*.

In Fig. 20 (a), we show the disk I/O of all the methods for TreeBank, which shows that the I/O cost of *PRIX* is obviously better than the XB-tree used by the other three methods. But our method is slightly better than *TwigStack* and *Twig*^{2}*Stack*. It is because by our method, not only the ancestor-descendant relationship, but the *left-to-right* order is also used to skip over entries in XB-trees.

However, due to the high *CPU* cost of *PRIX*, our method still outperforms *PRIX* in total query process time, as demonstrated in Fig. 20 (b). The reason for this is the multiple appearance of the same tag names, leading to a huge number of checkings of tree embedding as analyzed in the introduction. In Fig. 21, we show the time for I/O, which demonstrates that the I/O occupies only a very small fraction of the whole process time.



(a) (b)

Fig. 20. I/O page access and execution time for TreeBank.

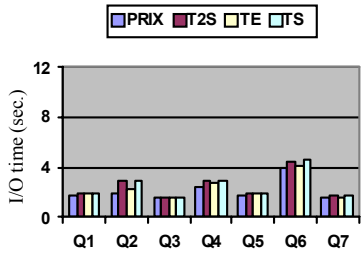
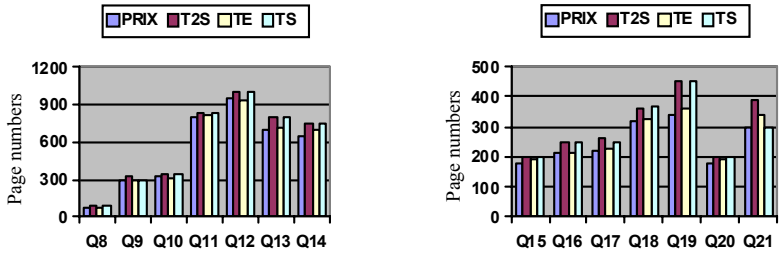


Fig. 21. I/O time for TreeBank.



(a) (b)

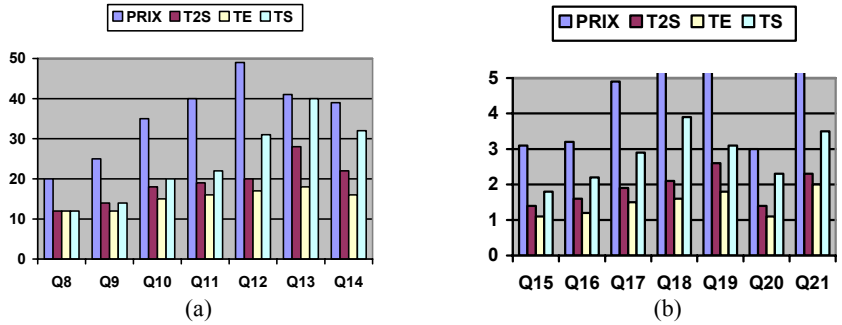
Fig. 22. I/O page access for DBLP and XMark.

In Fig. 22 (a), we show the I/O costs for the DBLP.

From this we can see that for Q_8 our method has less disk access than *PRIX* since most solutions were clustered in small region of the input stream, allowing ours to quickly find the relevant pages. But for Q_9, ours and *PRIX* have comparable I/O costs. For Q_{10}, ours works better. It is because the element *key* occurred in every document in the DBLP dataset and not much filtering can be achieved by *PRIX*. But ours can still skip a lot of leaf nodes by using the *ancestor-descendant* as well as *left-to-right* relationships. Q_{11} and Q_{12} are two quite low selectivity queries. Again, ours and *PRIX* have comparable I/O costs for them. Q_{13} and Q_{14} are also two quite low selectivity queries. But the tree matchings occur in different palces in document trees from Q_{11} and Q_{12}, and no significant difference between ours and *PRIX* can be observed. For this group of queries, both *TwigStack* and *Twig*^{2}*Stack* are generally worse than ours and *PRIX*. However, the difference is small. In fact, both the trie structure used by *PRIX* and the XB-tree used by the other three methods use the same information to filter data: label equality and the tree structural information. They have the same filtering power.

In Fig. 22 (b), we show the I/O costs for the XMark (with the scaling factor = 1), for which *PRIX* and the XB-tree are comparable.

In Figs. 23 (a) and (b), we show the whole execution times for processing queries against DBLP and XMark, respectively. From these, we can see that *PRIX* is much worse than all the other three methods. In fact, for both the DBLP and the XMark, *PRIX* exhibits an exponential time behavior. Although in the DBLP each path in a document tree is short and possesses no recursive structure, each Prüfer sequence (representing a document tree) has always multiple appearance of the same tag names. Each may lead to a checking of



(a) (b)

Fig. 23. Execution time for DBLP and XMark.

tree embedding. But most of them are not successful. The same analysis applies to the XMark.

In Figs. 24 (a)-(d), we show the scalability experimental results for XMark with different document sizes. We vary the XMark scale factor from 2 to 5. As can be seen, *PRIX* and *TwigStack* grow much faster than ours and *Twig*^{2}*Stack* in terms of the document size. Our method again has the best execution time.

Finally, to observe the impact of XB-trees, we have run our algorithm without indexes for DBLP and recorded the number of page access, shown in Table 6.

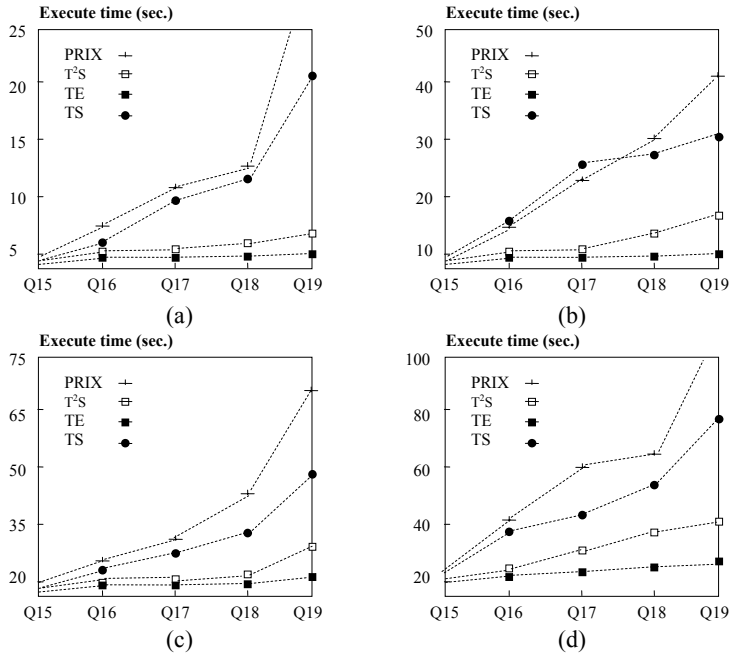


Fig. 24. Test results for XMark with different scaling factors.

Table 6. Disk I/O – DBLP.

Query	TE without XB	TE
Q8	1715 pages	93 pages
Q9	3036 pages	310 pages
Q10	4804 pages	401 pages
Q11	5563 pages	804 pages
Q12	6409 pages	976 pages
Q13	5230 pages	782 pages
Q14	5140 pages	745 pages

From this, we can see that using the XB-tree the I/O costs are effectively reduced. The *tree-embedding* algorithm examines every element in the sorted input streams while *XB-tree-embedding* uses XB-trees to skip elements in the sorted data streams. Especially, for

the ordered tree embedding, we are able to save a lot of leaf node checking by using *ancestor-descendant* as well as *left-to-right* relationships during an XB-tree search.

6. RELATED WORK

Besides the strategies for processing the ordered tree pattern queries (which are reviewed in the introduction), there is much research on the unordered tree pattern matching. Nearly all the proposed strategies can roughly be divided into two categories. One is index-based and the other is for the so-called XML streaming environment. For example, the methods discussed in [2, 17, 23-26, 32, 42] are typically index-based, by which a document is decomposed into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations, or into a set of paths. The sizes of intermediate relations tend to be very large, even when the input and final result sizes are much more manageable. As an important improvement, *TwigStack* was proposed by Bruno *et al.* [4], 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 [12]). This method is further improved by several researchers. In [8], *iTwigJoin* was discussed, which exploits different data partition strategies. In [24], *TJFast* accesses only leaf nodes by using extended Dewey IDs. By both methods, however, the path joins cannot be avoided. The method *Twig*^{2}*Stack* proposed by Chen *et al*. [12] works in a quite different way. It represents tree results using the so-called *hierarchical stack encoding* to avoid any possible useless path matchings. In [12], it is claimed that *Twig*^{2}*Stack* needs only $O(|D| \cdot |Q| + |subTwigResults|)$ time for generating paths. But a careful analysis shows that the time complexity for this task is actually bounded by $O(|D| \cdot |Q|^2 + |subTwigResults|)$. 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 [12] to know the working process). The following example helps for explanation.

In Fig. 25 (b), we show the hierarchical stacks associated with the two nodes *A* and *B* of *Q* with respect to *T* shown in Fig. 25 (a). In [12], 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

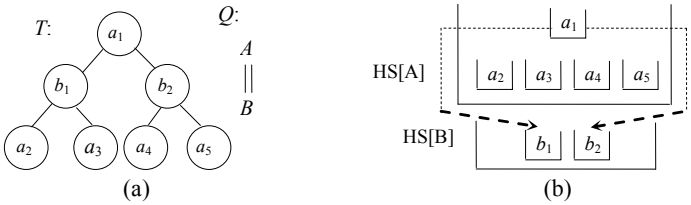


Fig. 25. Illustration for hierarchical stacks.

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 a $O(|D| \cdot |Q|^2)$ time complexity. The method discussed in [28] improves the stack structure used in *Twig*^{2}*Stack* to avoid storing individual path matches and remove *subTwigResults* time. But its theoretical time complexity is still $O(|D| \cdot |Q|^2)$. The method discussed in [20] also needs $O(|D| \cdot |Q|^2)$ time although some checkings can be saved by using ancestor/descendant relationships (see Property 1 on page 854). The above problem of *Twig*^{2}*Stack* is not removed.

A large amount of work has also been done on unordered tree pattern matching in an XML streaming environment, such as the methods discussed in [11, 16, 29, 44]. The time complexity of the method proposed in [11] is bounded by $O(T_h Q_d |Q||T| + |Q|^2|T|)$, where T_h is the height of T and Q_d is the largest outdegree of a node in Q. Both the methods discussed in [16, 29] require only $O(|Q||T|)$ time. However, by the method discussed in [29], extra value joins are needed. The algorithm described in [44] checks whether a document T embeds a query Q, returning a boolean value. Its time and space complexities are bounded by $O(|Q| \cdot \log|Q| \cdot r \cdot \log T_h)$ and $O(|T| \cdot |Q| \cdot \log|Q| \cdot r)$, respectively, where r is the recursion depth of T, which is defined to be the length of a longest path in T, whose start node and the end node are of the same label. For all these strategies, the order between siblings is not considered.

Finally, we point out that the bottom-up tree matching was first proposed in [19]. But it concerns a very strict tree matching, by which the matching of an edge to a path is not allowed. In [18], Gottlob *et al*. identified an XPath fragment called Core XPath, which can be evaluated in $O(|T| \cdot |Q|)$ time. Core XPath is slightly more expressive than the tree pattern queries in that it includes axes other than /-edges and //-edges. However, algorithms in [18] cannot be modified to use index structures since they require scanning XML documents in multiple passes. In [26], an algorithm for *tree homomorphism* is discussed. As with the algorithm proposed in [44], it examines whether a tree contains another and returns a boolean answer. But our algorithms show all the subtrees that are able to embed a given tree pattern. The node selecting Queries considered in [22] are in fact a kind of extended containment queries (whether a tree contains a certain node [42]) and cannot be used for the general purpose of twig joins. In [43], a special kind of tree matching, called *tree homeomorphism*, is discussed, which looks for a mapping that maps each edge in Q to a path in T.

7. 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 complexities of the algorithm is bounded by $O(|D| \cdot |Q| + |T| \cdot leaf_Q)$ and its space overhead is by $O(leaf_T \cdot leaf_Q)$, where T stands for a document tree, Q for a tree pattern and D is a largest data stream associated with a node q of Q, which contains the database nodes that match the node predicate at q. $leaf_T$ ($leaf_Q$) represents the number of the leaf nodes of T (resp. Q). The algorithm can also be implemented with the *XB*-tree index being used. Our experiments demonstrate that our method is both effective and efficient for the evaluation of ordered tree pattern queries.

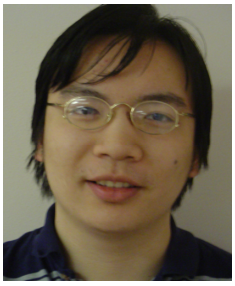
REFERENCES

1. S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publisher, Los Altos, USA, 1999.
2. S. A. Aghili, H. G. Li, D. Agrawal, and A. E. Abbadi, "TWIX: Twig structure and content matching of selective queries using binary labeling," in *Proceedings of the 1st International Conference on Scalable Information Systems*, 2006, pp. _____.
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 *Proceedings of IEEE International Conference on Data Engineering*, 2002, pp. _____.
4. N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: Optimal XML pattern matching," in *Proceedings of SIGMOD International Conference on Management of Data*, 2002, pp. 310-321.
5. B. Catherine and S. Bird, "Towards a general model of Interlinear text," in *Proceedings of Electronic Metastructure for Endangered Language Data Workshop*, 2003, pp. _____.
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. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML query language for heterogeneous data sources," in *Proceedings of the 3rd International Workshop on the Web and Databases*, 2000, pp. _____.
8. T. Chen, J. Lu, and T. W. Ling, "On boosting holism in XML twig pattern matching," in *Proceedings of SIGMOD*, 2005, pp. 455-466.
9. B. Choi, M. Mahoui, and D. Wood, "On the optimality of holistic algorithms for twig queries," in *Proceedings of International Conference Database and Expert Systems Applications*, 2003, pp. 235-244.
10. C. Chung, J. Min, and K. Shim, "APEX: An adaptive path index for XML data," *ACM SIGMOD*, 2002, pp. _____.
11. Y. Chen, S. B. Davison, and Y. Zheng, "An efficient XPath query processor for XML streams," in *Proceedings of International Conference on Data Engineering Workshops*, 2006, pp. _____.
12. 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 *Proceedings of International Conference on Very Large Data Bases*, 2006, pp. 283-294.
13. B. F. Cooper, N. Sample, M. Franklin, A. B. Hialtason, and M. Shadmon, "A fast index for semistructured data," in *Proceedings of International Conference on Very Large Data Bases*, 2001, pp. 341-350.
14. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A query language for XML," in *Proceedings of the 8th World Wide Web Conference*, 1999, pp. 77-91.
15. D. Florescu and D. Kossman, "Storing and querying XML data using an RDMBS," *IEEE Data Engineering Bulletin*, Vol. 22, 1999, pp. 27-34.
16. G. Gou and R. Chirkova, "Efficient algorithms for evaluating XPath over streams," in *Proceedings of SIGMOD*, 2007, pp. _____.
17. R. Goldman and J. Widom, "DataGuide: Enable query formulation and optimization in semistructured databases," in *Proceedings of International Conference on Very*

Large Data Bases, 1997, pp. 436-445.
18. G. Gottlob, C. Koch, and R. Pichler, "Efficient algorithms for processing XPath queries," *ACM Transactions on Database Systems*, Vol. 30, 2005, pp. 444-491.
19. C. M. Hoffmann and M. J. O'Donnell, "Pattern matching in trees," *Journal of ACM*, Vol. 29, 1982, pp. 68-95.
20. Z. Jiang, C. Luo, W. C. Hou, Q. Zhu, and D. Che, "Efficient processing of XML twig pattern: A novel one-phase holistic solution," in *Proceedings of the 18th International Conference on Database and Expert Systems Applications*, 2007, pp. 87-97.
21. R. Kaushik, P. Bohannon, J. Naughton, and H. Korth, "Covering indexes for branching path queries," in *Proceedings of ACM SIGMOD*, 2002, pp. _____.
22. C. Koch, "Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach," in *Proceedings of International Conference on Very Large Data Bases*, 2003, pp. _____.
23. Q. Li and B. Moon, "Indexing and querying XML data for regular path expressions," in *Proceedings of International Conference on Very Large Data Bases B*, 2001, pp. 361-370.
24. 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 *Proceedings of International Conference on Very Large Data Bases*, 2005, pp. 193-204.
25. J. McHugh and J. Widom, "Query optimization for XML," in *Proceedings of International Conference on Very Large Data Bases*, 1999, pp. _____
26. G. Miklau and D. Suciu, "Containment and equivalence of a fragment of XPath," *Journal of ACM*, Vol. 51, 2004, pp. 2-45.
27. K. Müller, "Semi-automatic construction of a question treebank," in *Proceedings of the 4th International Conference on Language Resources and Evaluation*, 2004, pp. _____.
28. L. Qin, J. X. Yu, and B. Ding, "TwigList: Make twig pattern matching fast," in *Proceedings of the 12th International Conference on Database Systems for Advanced Applications*, 2007, pp. 850-862.
29. P. Ramanan, "Holistic join for generalized tree patterns," *Information Systems*, Vol. 32, 2007, pp. 1018-1036.
30. P. Rao and B. Moon, "Sequencing XML data and query twigs for fast pattern matching," *ACM Transactions on Database Systems*, Vol. 31, 2006, pp. 299-345.
31. 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.
32. C. Seo, S. Lee, and H. Kim, "An efficient index technique for XML documents using RDBMS," *Information and Software Technology*, Vol. 45, 2003, pp. 11-22.
33. 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 *Proceedings of International Conference on Very Large Data Bases*, 1999, pp. _____.
34. University of Washington, "The Tukwila system," http://data.cs.washington.edu/integration/tukwila/.
35. University of Wisconsin, "The Niagara system," http://www.cs.wisc.edu /niagara/.
36. University of Washington, "XML repository," http://www.cs.washington.edu/research/

xmldatasets.
37. H. Wang, S. Park, W. Fan, and P. S. Yu, "ViST: A dynamic index method for querying XML data by tree structures," in *Proceedings of SIGMOD International Conference on Management of Data*, 2003, pp. _____.
38. H. Wang and X. Meng, "On the sequencing of tree structures for XML indexing," in *Proceedings of Conference on Data Engineering*, 2005, pp. 372-385.
39. World Wide Web Consortium, XML Path Language (XPath), W3C Recommendation, 2007, http://www.w3.org/TR/xpath20.
40. World Wide Web Consortium, XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, 2007, http://www.w3.org/TR/xquery.
41. "XMARK: The XML-benchmark project," http://monetdb.cwi.nl/xml, 2002.
42. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in *Proceedings of ACM SIGMOD*, 2001, pp. _____.
43. M. Götz, C. Koch, and W. Martens, "Efficient algorithms for the tree homeomorphism problem," in *Proceedings of International Symposium on Database Programming Language*, 2007, pp. _____.
44. Z. Bar-Yossef, M. Fontoura, and V. Josifovski, "On the memory requirements of XPath evaluation over XML streams," *Journal of Computer and System Sciences*, Vol. 73, 2007, pp. 391-441.
45. R. B. Lyngs, M. Zuker, and C. N. S. Pedersen, "Internal loops in RNA secondary structure prediction," in *Proceedings of the 3rd annual international conference on computational molecular biology*, 1999, pp. 260-267.
46. 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*, Vol. 7, 1999, pp. 359-368.
47. M. Zaki, "Efficiently mining frequent trees in a forest," in *Proceedings of International Conference on Knowledge Discovery in Databases*, 2002, pp. _____.

Yangjun Chen (陳____) received his B.S. in Information System Engineering from the Technical Institute of Changsha, China, in 1982, and his Diploma and Ph.D. in Computer Science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as a Post-Doctor at the Technical University of Chemnitz-Zwickau, Germany. After that, he worked as a Senior Engineer at the German National Research Center of Information Technology (GMD) for more than two years. Since 2000, he has been a Professor in the Department of Applied Computer Science at the University of Winnipeg, Canada. His research interests include deductive databases, federated databases, document databases, constraint satisfaction problem, graph theory and combinatorics. He has more than 150 publications in these areas.



Yibin Chen (陳____) received his B.S. and master degree from the Department of Electrical and Computer Engineering, University of Waterloo, and in the Department of Electrical and Computer Engineering, University of Toronto, Canada, respectively. Now he is a software engineer.