Document Tree Reconstruction and Fast Twig Pattern Matching

Yangjun Chen

Dept. Applied Computer Science, University of Winnipeg

515 Portage Ave., Winnipeg, Manitoba, Canada R3B 2E9

y.chen@uwinnipeg.ca

ABSTRACT

In this article, we discuss an efficient algorithm for tree mapping problem in XML databases. Given a target tree *T* and a pattern tree *Q*, the algorithm can find all the embeddings of *Q* in *T* in O(|*D*||*Q*|) time, where *D* is a largest data stream associated with a node of *Q*. Furthermore, the algorithm can be easily adapted to an indexing environment with *XB*-trees being used.

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

1. INTRODUCTION

In XML [31, 32], data are represented as a tree; associated with each node of the tree is an element name tag from a finite alphabet ∑. The children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element.

To abstract from existing query languages for XML (e.g. XPath [14], XQuery [32], XML-QL [13], and Quilt [5, 6]), we express queries as twig (small tree) patterns, where nodes are labeled with symbols from ∑ ∪ {*} (* is a wildcard, matching any node name) and string values, and edges are *parent-child* or *ancestor-descendant* relationships. As an example, consider the query tree shown in Fig. 1.

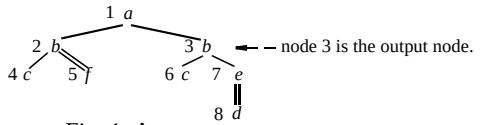

Fig. 1. A query tree

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

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

In this figure, there are two kinds of edges: child edges (/-edges for short) for parent-child relationships, and descendant edges (//-edges for short) for ancestor-descendant relationships. A /-edge from node *v* to node *u* is denoted by $v \rightarrow u$ in the text, and represented by a single arc; *u* is called a */-child* of *v*. A //-edge is denoted $v \Rightarrow u$ in the text, and represented by a double arc; *u* is called a *//-child* of *v*.

In any DAG (*directed acyclic graph*), a node *u* is said to be a descendant of a node *v* if there exists a path (sequence of edges) from *v* to *u*. In the case of a twig pattern, this path could consist of any sequence of /-edges and/or //-edges. We also use *label*(*v*) to represent the symbol (∈ ∑ ∪ {*}) or the string associated with *v*. Based on these concepts, the tree embedding can be defined as follows.

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

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

(ii) Preserve *parent-child*/*ancestor-descendant* relationships: 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*.

Notice that an embedding could map several nodes of the query (of the same type) to the same node of the database. It also allows a tree mapped to a path. This definition is quite different from the tree matching defined in [16].

There is much research on how to find such a mapping efficiently and all the proposed methods can be categorized into two groups. By the first group [1, 9, 11, 14, 19, 22, 28, 29, 32, 33, 34], a tree pattern is typically decomposed into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations. Then, an index structure is used to find all the matching pairs that are joined together to form the final result. By the second group [4, 7, 8, 10, 18, 20], a query pattern is decomposed into a set of paths. The final result is constructed by joining all the matching paths together. For all these methods, the join operations involved require exponential time in the worst case. For example, if we decompose a twig pattern into paths to find all the matching paths from a database, we need $O(p^{\lambda})$ time to join them together, where *p* is the largest length of a matching path and λ is the number of all such paths.

In this paper, we proposed a new algorithm with no join operations involved. The algorithm runs in $O(|T| \cdot Q_{leaf})$ time and $O(T_{leaf} Q_{leaf})$ space, where T_{leaf} and Q_{leaf} represent the numbers of the leaf nodes in *T* and in *Q*, respectively.

In this paper, we present an new algorithm, *tree-matching*( ), for evaluating twig pattern queries and adapt it into an environment with the following advantages:

- *tree-matching*( ) is able to handle twig patterns containing /-edges, //-edges, *, and branches.
- *tree-matching*( ) runs in O(|*D*|·|*Q*|) time and O(|*D*|·|*Q*|) space, where *D* is a largest data stream associated with a node of *Q*.
- *tree-matching*( ) generates neither matching paths nor hierarchical stacks [11]. Therefore, the costly path joins [2, 4], or join-like operations (such as the *result enumeration* used in [11]), are unnecessary.

In fact, the path join used in [4] leads to an exponential time complexity for queries containg both /-edges and //-edges (as shown in [11]) while the result enumberation used in [11] can be completely avoided by our method.

The remainder of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we restate the tree encoding [34], which facilitates the recognition of different relationships among the nodes of a tree. In Section 4, we discuss our algorithm, which is adapted to an index environment in Section 5. Section 6 is devoted to the implementation and experiments. Finally, a short conclusion is set forth in Section 7.

2. TREE ENCODING

In [34], an interesting tree encoding method was discussed, which can be used to identify different relationships among the nodes of a tree.

Let T be a document tree. We associate each node v in T with a quadruple (*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. 3 for illustration.) By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined [34]:

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

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

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

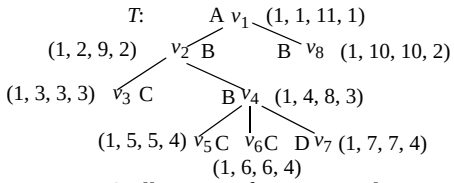

Fig. 3. Illustration for tree encoding

In Fig. 3, 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 $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$, 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 use v and $\alpha(v)$ interchangeably.

We can also assign LeftPos and RightPos values to the query nodes in Q for the same purpose as above.

Finally we use $T[v]$ to represent a subtree rooted at v in T.

3. MAIN ALGORITHM

In this section, we discuss our algorithm according to Definition 1. The main idea of this algorithm is the so-called *subtree reconstruction*, by which a tree structure is established according to a given set of quadruples (called a data stream in [4]). Therefore, we will first discuss an algorithm for this task in 3.1. Then, in 3.2, we give our algorithm to check twig patterns that contains /-edges, //-edges, * and branches.

3.1 Tree reconstruction

As with *TwigStack* [4], each node q in a twig 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. 4, we show a query tree containing 5 nodes and 4 edges and each node is associated with a list of matching nodes of the document tree shown in Fig. 3, sorted according to their (DocID, LeftPos) values. For simplicity, we use the node names in a list, instead of the node's quadruples.

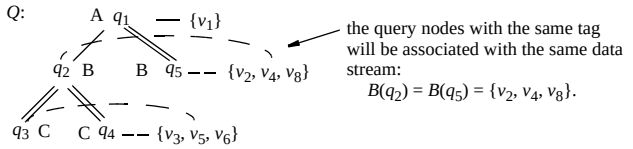

Fig. 4. Illustration for $L(q_i)$'s

Note that iterating through the stream nodes in sorted order of thier LeftPos values corresponds to access of document nodes in preorder. However, our algorithm needs to visit them in *postorder* (i.e., in 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 them at the end of the corresponding $L(q_i)$'s (see lines 3 - 4); and push v into *ST* (see line 7.) The output of the algorithm is a set of data streams $L(q_i)$'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. 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. 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 $L(Q) = \{L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)\}$ 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.

First, we discuss how to reconstruct a treeFirst, we discuss how to reconstruct a tree structure from 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 children of u. (See Fig. 5.)

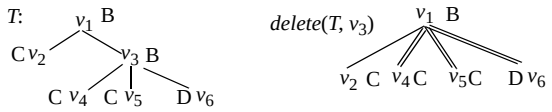

Fig. 5. The effect of removing v_3 from T

Definition 2. (*matching subtrees*) A matching subtree T' of T with respect to a twig 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. 6(a) is a matching subtree of the document tree shown in Fig. 3 with respect to the query tree shown in Fig. 6(b).

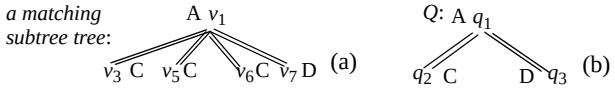

Fig. 6. A matching tree and a query tree

Given $L(Q)$, what we want is to construct a matching subtree from them to facilitate the checking of twig 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 created, let v' be the node chosen just before v. Then, the following will be performed.
   (i) 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'.
   (ii) 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. Set *left-sibling*(v) ← v''.

Fig. 7 is a pictorial illustration of this process.

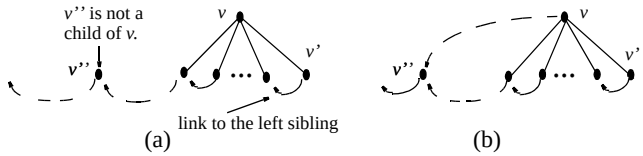

Fig. 7. Illustration for the construction of a matching subtree

In Fig. 7(a), we show the navigation along a left-sibling chain starting from v' when we find that v' is a child (descendant) of v. This process stops whenever we meet v'', a node that is not a child (descendant) of v. Fig. 7(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 leftmost child.

Below is a 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 unordered twig 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(q) in L(Q) become empty
2.  {identify q such that the first element v of L(q) is of the minimal
      RightPos value; remove v from L(q);
3.  generate node v;
4.  if v is not the first node created then
5.  { let v' be the node generated just before v;
6.    if v' is not a child (descendant) of v then
7.    {left-sibling(v) ← v';}(*generate a left-sibling link.*)
8.    else
9.    { v'' ← v'; w ← v';    (*v'' and w are two temporary
                                variables.*)
10.     while v'' is a child (descendant) of v do
11.     { parent(v'') ← v; (*generate a parent link. Also,
                  indicate whether v'' is a /-child or a //-child.*)
12.       w ← v''; v'' ← left-sibling(v'');
13.     }
14.     left-sibling(v) ← v'';
15.   }
16. }
```

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 7). 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 9 - 13.) Finally, the left-sibling link of v is set to be v'' (see line 14).

Example 1. Consider the twig pattern shown in Fig. 4 once again, in which we 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. 8.

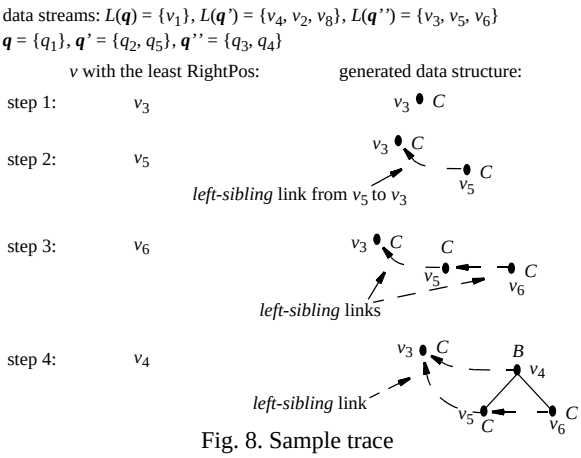

Fig. 8. Sample trace

In step 1 (see Fig. 8), 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 enables us to reconstruct a matching subtree in an efficient way.

The subsequence computation is shown in Fig. 9. □

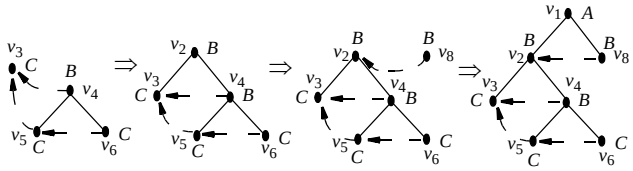

Fig. 9. Sample trace

Proposition 1. Let T be a document tree. Let Q be a twig pattern. Let $L(Q) = \{L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)\}$ 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 $L = |L(\boldsymbol{q}_1)| + ... + |L(\boldsymbol{q}_l)|$. 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 $\{u_1, ..., u_k, u_{k+1}\}$ 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 $\{u_1, ..., u_k\}$. 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}, we consider two cases:

i) v_{k+1} is an ancestor of v_k,

ii) v_{k+1} is to the right of v_k.

In case (i), the algorithm will generate an edge (v_{k+1}, v_k), and then travel 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 (ii), 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 Twig pattern matching

In fact, the algorithm discussed in 4.1 hints an efficient way for twig pattern matching.

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* $QS(v)$. That is, each time we choose a v with the largest LeftPos value from a data stream $L(\boldsymbol{q})$, we will insert all the query nodes in $\boldsymbol{q}$ into $QS(v)$. For example, in the first step shown in Fig. 9, the query node stream for v_8 can be determined as shown in Fig. 10(a).

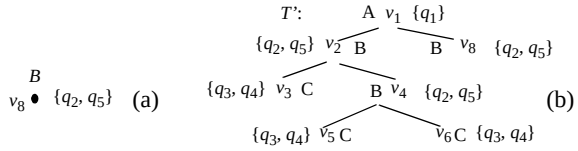

Fig. 10. Illustration for generating QS's

In this way, we can create a matching subtree as illustrated in Fig. 10(b), in which each node in T' is associated with a sorted query node stream. If we check, before a q is inserted into the corresponding $QS(v)$, whether $Q[q]$ (the subtree rooted at q) can be imbedded into $T'[v]$, we get in fact an algorithm for twig pattern matching. The challenge is how to conduct such a checking efficiently.

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

i) v is the node currently encountered.

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

iii) q is a //-child, or
q is a /-child, and u is a /-child with $label(u) = label(q)$.

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

1. Let $q_1, ..., q_k$ be the child nodes of q.
2. If for each q_i $(i = 1, ..., k)$, $\chi(q_i)$ is equal to v and $label(v) = label(q)$, insert q into $QS(v)$.

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

Let $v_1, ..., v_j$ be the children of v in T'. All the $QS(v_i)$'s $(i = 1, ..., j)$ should also be added into $QS(v)$. This process can be elaborated as follows.

Let $QS(v_i) = \{ q_{i_1}, ..., q_{i_l} \}$ $(i = 1, ..., j)$. Then, we have q_{i_1}.LeftPos < ... < q_{i_l}.LeftPos. (Recall that all the query nodes inserted into $QS(v_i)$ come from a same $\boldsymbol{q}$, in which all the elements are sorted by their LeftPos values.) Each time we insert a q into $QS(v_i)$, we can check whether it is subsumed by the query node q' which has just been inserted before. If it the case, q will not be inserted since the embedding of $Q[q']$ in $T[v_i]$ implies the embedding of $Q[q]$ in $T[v_i]$. (Note that LeftPos(q') < LeftPos(q), q cannot be an ancestor of q'.) Thus, $QS(v_i)$ contains only those query nodes which are not on the same path. Therefore, we must also have q_{i_1}.RightPos < ... < q_{i_l}.RightPos. So the query nodes in $QS(v_i)$ are increasingly sorted by both LeftPos and RightPos values. Obviously, $|QS(v_i)| \le leaf_Q$. We can store $QS(v_i)$ as a linked list. Let QS_1 and QS_2 be two sorted lists with $|QS_1| \le leaf_Q$ and $|QS_2| \le leaf_Q$. The union of QS_1 and QS_2 $(QS_1 \cup QS_2)$ can be performed by scanning both QS_1 and QS_2 from left to right and inserting the query node of QS_2 into QS_1 one by one. During this process, any query node in QS_1, which is subsumed by some query node in QS_2 will be removed; and any query node in QS_2, which is subsumed by some query in QS_1, will not be inserted into QS_1. The result is stored in QS_1. From this, we can see that the

resulting linked list is still sorted and its size is bounded by $leaf_Q$. We denote this process as *merge*(QS_1, QS_2) and define *merge*(QS_1, ..., QS_{j-1}, QS_j) to be *merge*(*merge*(QS_1, ...QS_{j-1}), QS_j).

In the following, we present our first algorithm *A1-1*($L(Q)$) for queries containing only /-edges, //-edges, and branches. During the process, another algorithm *subsumption-check*(v, $\boldsymbol{q}$) may be invoked to check whether any $q \in \boldsymbol{q}$ can be inserted into $QS(v)$, where $\boldsymbol{q}$ is a subset of query nodes such that $L(\boldsymbol{q})$ contains v.

The algorithm *A1-1*($L(Q)$) is similar to Algorithm *matching-tree-construction*( ), by which a quadruple is removed in turn from the data stream and a node v for it is generated and inserted into the matching subtree.

In addition, two data structures are used:

D_{root} - a subset of document nodes v such that Q can be embedded in $T[v]$.

D_{output} - a subset of document nodes v such that $Q[q_{output}]$ can be embedded in $T[v]$, where q_{output} is the output node of Q.

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

Algorithm *A1-1*($L(Q)$)
input: all data streams $L(Q)$.
output: a matching subtree T' of T, D_{root} and D_{output}.
begin
1. **repeat until** each $L(\boldsymbol{q})$ in $L(Q)$ becomes empty {
2. identify $\boldsymbol{q}$ such that the first node v of $L(\boldsymbol{q})$ is of the minimal RightPos value; remove v from $L(\boldsymbol{q})$; generate node v;
3. **if** v is the first node created **then**
4. {$QS(v) \leftarrow$ *subsumption-check*(v, $\boldsymbol{q}$);}
5. **else**
6. { let v' be the quadruple chosen just before v, for which a node is constructed;
7. **if** v' is not a child (descendant) of v **then**
8. { *left-sibling*(v) $\leftarrow v'$;
9. $QS(v) \leftarrow$ *subsumption-check*(v, $\boldsymbol{q}$);}
10. **else**
11. { $v'' \leftarrow v'$; $w \leftarrow v'$; (*v'' and w are two temporary units.*)
12. **while** v'' is a child (descendant) of v **do**
13. { *parent*(v'') $\leftarrow v$; (*generate a parent link. Also, indicate whether v'' is a /-child or a //-child.*)
14. **for** each q in $QS(v'')$ **do** {
15. **if** ((q is a //-child) or
16. (q is a /-child and v'' is a /-child and
17. *label*(q) = *label*(v'')))
18. **then** $\chi(q) \leftarrow v$;}
19. $w \leftarrow v''$; $v'' \leftarrow$ *left-sibling*(v'');
20. remove *left-sibling*(w);
21. }
22. *left-sibling*(v) $\leftarrow v''$;
23. }
24. $\boldsymbol{q} \leftarrow$ *subsumption-check*(v, $\boldsymbol{q}$);
25. let v_1, ..., v_j be the child nodes of v;
26. $\boldsymbol{q}' \leftarrow$ *merge*($QS(v_1)$, ..., $QS(v_j)$);
27. remove $QS(v_1)$, ..., $QS(v_j)$;
28. $QS(v) \leftarrow$ *merge*($\boldsymbol{q}$, $\boldsymbol{q}'$);
29. }}
end

Function *subsumption-check*(v, $\boldsymbol{q}$) (*v satisfies the node name test at each q in $\boldsymbol{q}$.*)
1. $QS \leftarrow \Phi$;
2. **for** each q in $\boldsymbol{q}$ **do** {
3. let q_1, ..., q_j be the child nodes of q.
4. **if** for each /-child q_i $\chi(q_i) = v$ and for each //-child q_i $\chi(q_i)$ is subsumed by v **then**
5. {$QS \leftarrow QS \cup \{q\}$;
6. **if** q is the root of Q **then**
7. $D_{root} \leftarrow D_{root} \cup \{v\}$;
8. **if** q is the output node **then** $D_{output} \leftarrow D_{output} \cup \{v\}$;}}
9. return QS;
end

The output of *A1-1*( ) is D_{root} and D_{output}. Based on them, we can generate another subtree T'' of T (like a matching subtree), which contains only those nodes v such that $T[v]$ contains $Q[r]$ with *label*(v) = *label*(r) or contains $Q[o]$ with *label*(v) = *label*(o), where r and o represent the root and the output node of Q, respectively. We call a node an *r-node* if $T[v]$ contains $Q[r]$ with *label*(v) = *label*(r), or an *o-node* if $T[v]$ contains $Q[o]$ 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 answer.

Algorithm *A1-1*( ) does almost the same work as Algorithm *matching-tree-construction*( ).The main difference is lines 14 - 18 and lines 24 - 28. In lines 14 - 18, we set χ values for some q's. Each of them appears in a $QS(v')$, where v' is a child node of v, satisfying the conditions i) - iii) given above. In lines 24 - 28, we use the merging operation to construct $QS(v)$.

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

Example 2. Applying Algorithm *A1-1* to the data streams shown in Fig. 4, we will find that the document tree shown in Fig. 3 contains the query tree shown in Fig. 4. We trace the computation process as shown in Fig. 11.

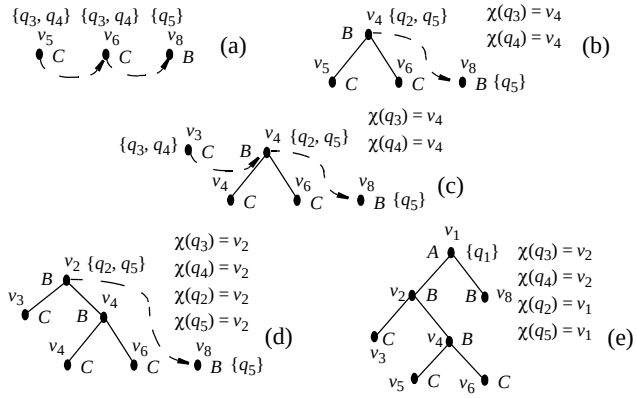

Fig. 11. Sample trace

In the first three steps, we will generate part of the matching subtree as shown in Fig. 11(a). Associated with v_8 is a query node stream: $QS(v_8) = \{q_5\}$. Although q_2 also matches v_8, it cannot survive the subsumption check (see line 4 in *subsumption-check*( )). So it does not appear in $QS(v_8)$. In addition, we have $QS(v_5) = QS(v_6) = \{q_3, q_4\}$. It is because both q_3 and q_4 are leaf nodes and can always satisfy the subsumption checking. In a next step, we will meet the parent v_4 (appearing in $L(\{q_2, q_5\})$ of v_5 and v_6. So we are able to get $\chi(q_3) = v_4$ and $\chi(q_4) = v_4$ (see Fig. 11(b)). In terms of these two values, we know that q_2 should be inserted into $QS(v_4)$. q_5 is a leaf node and also inserted into $QS(v_4)$. In addition, $QS(v_5)$ and $QS(v_6)$ should also be merged into it. In the fifth step, we meet v_3. $QS(v_3) = \{q_3, q_4\}$ (see Fig. 11(c)). In the sixth step, we meet v_2 (in $L(\{q_2, q_5\})$). It

is the parent of v_3 and v_4. According to $QS(v_3) = \{q_3, q_4\}$ and $QS(v_4) = \{q_2, q_5\}$, as well as the fact that both q_5 and v_4 are /-child nodes and $label(q_5) = label(v_4) = B$, we will set $\chi(q_3) = \chi(q_4) = \chi(q_2) = \chi(q_5) = v_2$ (see Fig. 11(d)). Thus, we have $QS(v_2) = \{q_2, q_5\}$. Finally, in step 7, according to $QS(v_2) = \{q_2, q_5\}$ and $QS(v_8) = \{q_5\}$, we will set $\chi(q_2) = v_1$ and $\chi(q_5) = v_1$ (see Fig. 11(e)), leading to the insertion of q_1 into $QS(v_1)$. □

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

In the following, we prove the correctness of this algorithm. First, we prove a simple lemma.

Lemma 1. Let v_1, v_2, and v_3 be three nodes in a tree with v_3.LeftPos < v_2.LeftPos < v_1.LeftPos. If v_1 is a descendant of v_3. Then, v_2 must also be a descendant of v_3.

Proof. We consider two cases: i) v_2 is to the left of v_1, and ii) v_2 is an ancestor of v_1. In case (i), we have v_1.RightPos > v_2.RightPos. So we have v_3.RightPos > v_1.RightPos > v_2.RightPos. This shows that v_2 is a descendant of v_3. In case (ii), v_1, v_2, and v_3 are on the same path. Since v_2.LeftPos > v_3.LeftPos, v_2 must be a descendant of v_3. □

We illustrate Lemma 1 by Fig. 12, which is helpful for understanding the proof of Proposition 2 given below.

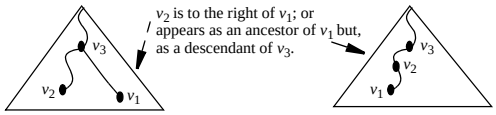

Fig. 12. A matching subtree with *QS*'s

Proposition 2. Let Q be a twig pattern containing only /-edges, //-edges and branches. Let v be a node in the matching subtree T' with respect to Q created by Algorithm *A1-1*. Let q be a node in Q. Then, q appears in $QS(v)$ if and only if $T'[v]$ contains $Q[q]$.

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

(1) $label(v) = label(q)$.

(2) For each //-child q' of q there exists a child v' of v such that $T[v']$ contains $Q[q']$. (See line 15 in *A1-1*( ).)

(3) For each /-child q'' of q there exists a /-child v'' of v such that $T[v'']$ contains $Q[q'']$ and $label(v'') = label(q'')$. (See lines 16 - 17 in *A1-1*( ).)

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

Only-if-part. The proof of this part is tedious. In the following, we give only a proof for the simple case that Q contains no /-edges, which is done by induction of the height h of the nodes in T'.

Basis. When $h = 0$, for the leaf nodes of T', the proposition trivially holds.

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

v'.LeftPos ≥ v_l.LeftPos > v.LeftPos.

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

Lemma 1 helps to clarify the only-if part of the above proof. In fact, it reveals an important property of the tree encoding, which enables us to save both space and time. That is, it is not necessary for us to keep all the values of $\chi(q_i)$, but only one to check the ancestor-descendant/parent-child relationship. Due to this property, the path join [4], as well as the result enumeration [11], can be completely avoided.

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

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

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

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

However, if we record only those parts of T', which contain the whole Q or the subtree rooted at the output node, the runtime memory usage must be much less than $O(|D| \cdot |Q|)$ for the following two reasons:

(i) The QS data structure for a node is removed once its parent node is created. So the space overhead is bounded by $O(|D| \cdot leaf_Q)$

(ii) During the whole process, the elements in the data streams are removed one by one.

Of course, if we want to record all those parts of T', which contain one or more parts of Q, we need $O(|D| \cdot |Q|)$ space to store all the re-

sults.

In the above discussion, we handle wildcards in the same way as any non-wildcard nodes. But a wildcard matches any tag name. Therefore, *L*(*) should contain all the nodes in *T*. However, as we can see in the next section, by using the *XB*-tree [], *L*(*) contains a much smaller set of nodes in *T*. In fact, during the whole process, each entry in an *XB*-tree is accessed only once along the nodes' postorder numbers. That is, for each node in *Q*, no matter whether it is a wildcard or not, we only check it against the nodes currently encountered. Thus, with the help of *XB*-trees, * can be handled in the same as a non-wildcard, causing no extra time complexity.

4. CONCLUSION

In this paper, two new algorithms A1 and A2 are discussed, according to two different definitions of tree embedding. By the first definition, we consider only the ancestor-descendant relationship among the nodes in a tree structure. By the second definition, not only the ancestor-descendant relationship but also the order of siblings are taken into account. Almost all the existing strategies are designed according to the first definition. We provide the second definition as an option in the case that the user wants to do so. Both A1 and A2 have the best worst-case time complexities. Especially, we show that for the twig pattern matching problem, neither the join nor the result enumeration (a join-like operation) is necessary. Our experiments demonstrate that our methods are both effective and efficient for the evaluation of twig 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, 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] D. D. Chamberlin, J.Clark, D. Florescu and M. Stefanescu. "XQuery1.0: An XML Query Language," http://www.w3.org/TR/query-datamodel/.

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

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

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

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

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

[11] 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.

[12] 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.

[13] A. Dutch, 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.

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

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

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

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

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

[25] 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.

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

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

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.