On the Stack Encoding and Twig Joins

Abstract The twig join, which is used to find all occurrences of a twig pattern in an XML database, is a core operation for XML query processing. A great many strategies for handling this problem have been proposed and can be roughly classified into two groups. The first group decomposes a twig pattern (a small tree) into a set of binary relaionships between pairs of nodes, such as parent-child and ancestor-descendant relations; and transforms a tree matching problem into a series of simple relation look-ups. The second group decomposes a twig pattern into a set of paths. Among all this kind of methods, the approach based on the so-called *stack encoding* is very interesting, which can represent in linear space a potentially exponential (in the number of query nodes) number of matching paths. However, the available processes for generating such compressed paths suffer substantial redundancy and can be greatly improved. In this paper, we analyze this method and show that the time complexities of path generation in its two main procedures: *PathStack* and *TwigStack* can be reduced from $O(m^2 \cdot n)$ to $O(m \cdot n)$, where m and n are the sizes of the query tree and document tree, respectively.

Key Words: XML databases, Trees, Paths, XML pattern matching, Twig join

1. Introduction

In XML [13, 14], data is represented as a tree; associated with each node of the tree is an element type 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.

To abstract from existing query languages for XML (e.g. XPath [13], XQuery [14], XML-QL [5], and Quilt [3, 4]), we express queries as tree patterns where nodes are types from $\sum \cup \{*\}$ (* is a wildcard, matching any node type) 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 type b that is a child of some node of type a. In addition, the b-node is the parent of some c-node and some e-node, as well as an ancestor of some d-node. The query corresponds to the following XPath expression:

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

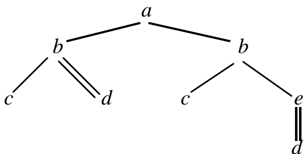

Fig. 1. A query tree

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

Finding all occurrences of a twig pattern in a database is a core operation in XML query processing, both in relational implementation of XML databases, and in native XML databases.

Recently this problem has received much attention in database research community and different strategies have been proposed [1, 2, 6, 7, 8, 9, 10, 11, 12, 15]. Most of them (for example, [1, 6, 7, 8, 9, 10, 15]) typically decompose a twig pattern into a set of binary relationships between pairs of nodes, such as parent-child and ancestor-descendant relations; and the sizes of intermediate relations tend to be very large, even when the input and final result sizes are much more manageable. Another kind of strategies bases on path decomposition, such as those discussed in [2, 11, 12]. In [11, 12], all the possible paths of an XML document are explicitly stored and indexed using B+-trees as well as trie structures. In [2], a document is also decomposed, but dynamically depending on the given queries. This method is of special interest since the decomposed paths are not simply stored but compressed by using the so-called *stack encoding*. Although the idea of compressing intermediate results is very attractive, the process suggested in [2] for producing compact paths is not so efficient and can be substantially improved.

In this paper, we analyze the method described in [2] and show its redundancy. In addition, two new algorithms are presented, which improve the two main pro-

cedures of this method: *PathStack* and *TwigStack*, by one order of magnitude.

The remainder of the paper is organized as follows. In Section 2, we analyze the first procedure *PathStack* proposed in [2], and present a new algorithm to improve its time complexity. In Section 3, we analyze the second procedure *TwigStack* discussed in [2], and show a way to reduce the time for path generation. In Section 4, we extend the method discussed in Section 3 to general cases. Finally, a short conclusion is set forth in Section 5.

2. Refined pathstack

In this section, we discuss the first procedure *PathStack* given in [2], which is used to evaluate a sort of simple queries that can be represented as a single path containing only *d*-edges. First, we describe the *PathStack* algorithm in 2.1. Then, we discuss how this algorithm can be improved in 2.2.

2.1 Description of PathStack

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. 2(a) for illustration.) By using such a data structure, the structural relationship between the nodes in an XML database can be determined easily [2]:

(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 $ln_1 = ln_2 + 1$.

(iii)f*rom 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 $d_1 = d_2$, $r_1 < l_2$.

Assume that $q = q_1 \Rightarrow q_2 \ldots \Rightarrow q_{m-1} \Rightarrow q_m$ be a path query. We associate each q_i ($1 \le i \le m$) with a data stream $L(q_i)$, which contains the quadruples of the database nodes that match q_i as illustrated in Fig. 2(b). Such a list can be established by using an efficient access mechanism, such as an index structure. In addition, the quadruples in a list are sorted by their (DocId, LeftPos) values.

The main idea of *PathStack* is to compress the matching paths using a set of stacks. Each of them is attached to a q_i ($1 \le i \le m$), denoted as $S(q_i)$, with the following properties:

(i) Each entry in $S(q_i)$ is a pair: (α, a pointer to an entry in $S(parent(q_i))$), where $\alpha \in L(q_i)$ is a quadruple for some node v.

(ii)The entries in $S(q_i)$ (from bottom to top) are guaranteed to lie on a root-to-leaf path in a document tree.

(iii)The set of stacks contain a compact encoding of partial and total answers to the query path.

To see how it works, let's have a look at the following example.

Example 1. In Fig. 2(a), we show a simple document T and a simple query q.

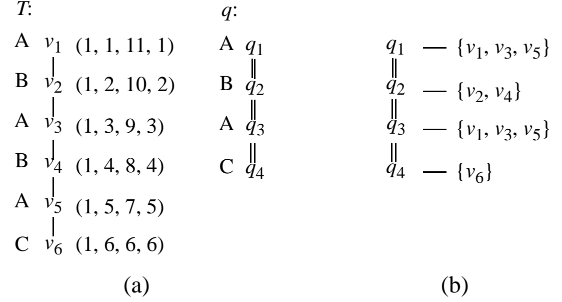

Fig. 2. A simple data setting

Obviously, T has four subpaths that match q, as shown in Fig. 3(a). By using the stack encoding, they can be stored in a way as shown in Fig. 3(b).

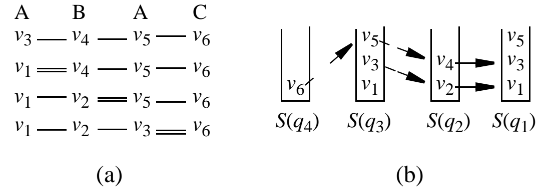

Fig. 3. Illustration for stack encoding

First, we notice that the answer [v_3, v_4, v_5, v_6] is encoded since v_6 points to v_5, v_5 to v_4, and v_4 to v_3. Also, the answer [v_1, v_4, v_5, v_6] is encoded since v_1 is below v_3 on the stack $S(q_1)$. For the same reason, [v_1, v_2, v_5, v_6] is an answer since v_2 is below v_4 on the stack $S(q_2)$ and has a pointer to v_1. Finally, since v_3 is below v_5 on the stack $S(q_3)$ and has a pointer to v_2, [v_1, v_2, v_3, v_6] is also an answer. However, [v_3, v_2, v_5, v_6] is not an answer since v_3 is above v_1 on $S(q_2)$, to which v_2 points. □

In the following, we will first describe *PathStack* given in [2] and analyze its time complexity. Then, we describe a new algorithm in the next subsection, which improves *PathStack* by one order of magnitude.

In *PathStack*, the following operations are used.

next($L(q_i)$): return the next element in $L(q_i)$. Initially, the pointer is to the position before the first element in $L(q_i)$.

advance($L(q_i)$): move to the next element in $L(q_i)$;

LeftPos(α): return the LeftPost of α;

RightPos(α): return the RightPost of α.

Algorithm *PathStack*(q)

1. **while** $\neg$ *end*(q) **do**
2. { $q_{min} \leftarrow$ *getMinSource*(q);
3. **for** each q_i in q **do**

4. **while** ($\neg$ empty($S(q_i)$) $\wedge$ (*RightPos*(top($S(q_i)$)) < *LeftPos*($L(q_{min})$)) **do** pop($S(q_i)$);
5. *moveStreamToStack*($L(q_{min})$, $S(q_{min})$, pointer to top(S(*parent*(q_{min}))));
6. **if** (q_{min} is a leaf node) **then**
7. {*showSolutions*($S(q_{min})$, 1); pop($S(q_{min})$);}
8. }

Function *end*(q)

if for any leaf node q', $L(q')$ is empty
then return *true*
else return *false*;

Function *getMinSource*(q)

return q_i in q such that *LeftPos*(*next*($L(q_i)$)) is minimal.

Procedure *moveStreamToStack*(L, S, p)

1. push(S, *next*(L), p);
2. *advance*(L);

The algorithm *PathStack* repeatedly construct stack encodings of partial and total answers by iterating through the streams associated with the nodes in q, which are in the order of sorted LeftPos values. So the nodes in T are checked in the order of non-decreasing LeftPos values. This is done by executing line 2, which invokes the procedure *getMinSources*(q) to identify the stream containing the next node to be processed. By executing lines 3 -5, some partial answers are removed from the corresponding stacks, which cannot be extended to total answers in terms of the ancestor-descendant relationships of nodes. Line 6 arguments the partial answers encoded in the stacks with new stream nodes. Whenever a node q_{min} is pushed onto the stack $S(q_{min})$ and that node is the leaf of the path query, the stack must have an encoding of some total answers if any. In this case, the algorithm *showSolutions* will be invoked to output these answers. Each of them is represented as an n-tuple that is in sorted leaf-to-root of the query path [2].

Procedure *showSolutions*(a, b)

1. index[a] $\leftarrow b$;
2. **if** ($a = 1$) **then**
3. output($S(q_n)$.index[n], ..., $S(q_1)$.index[1])
4. **else**
5. {**for** i = 1 to $S(q_a)$.index[a].*pointer-to-parent* **do**
6. *showSolution*(a - 1, i);}

The above algorithm expands the paths from the corresponding stack encodings. Assume that the nodes in the query path is numbered from top to bottom. We maintain a global array index[1..n], in which ith entry is a pointer to the position in $S(q_i)$ that we are interested in for the current solution, where the bottom of $S(q_i)$ is position 1.

The time complexity of *PathStack* can be estimated as follows. Let n_i be the size of $L(q_i)$. Then, the main **while**-loop will be iterated $\sum_{i=1}^{m} n_i$ times since the termination condition of this **while**-loop is when all the elements in $L(q_m)$ are exhausted. In each iteration, the top elements of m stacks are checked. Let δ_{ijk} be the number of elements removed from $S(q_k)$ in the (i, j)-th iteration. Then, the worst-case cost is bounded by

$$O\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}(1+\delta_{ijk})\right)$$

$$= O\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}1\right) + O\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}\delta_{ijk}\right)$$

$$= O\left(\sum_{i=1}^{m} m \cdot n_i\right) + O(m \cdot n) = O(m^2 \cdot n).$$

Here we should remark that $\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}\delta_{ijk}$ cannot be larger than $m \cdot n$ since at most $m \cdot n$ elements may be pushed on to the stacks.

2.2 Removing redundancy from PathStack

An observation shows that each time when *getMinSource*(q) is carried out, a database node is visited and all the database nodes are accessed in the order of non-decreasing LeftPos values. So we can rearrange the computation as below.

Definition 1 (*matching subtrees*) A matching subtree T' of T w.r.t a query path q (containing only d-edges) is a tree, in which each node matching the predicate at a node in q and there is an edge from node a to node b iff there exists a path p from a to b in T and any other node on p does not match any node predicate in q. □

Example 2. Consider the document tree T shown in Fig. 4(a). With respect to the query path q shown in Fig. 2(a), it has a matching subtree as shown in Fig. 4(b).

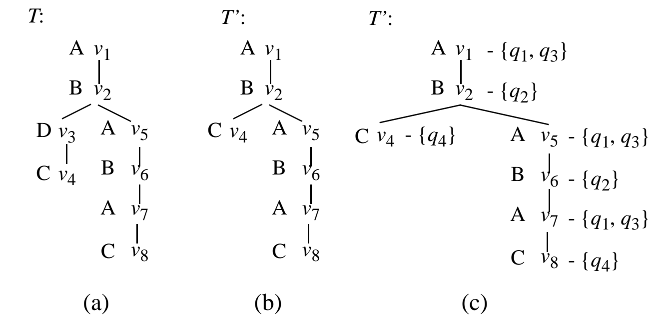

Fig. 4. Illustration for matching subtrees

The main idea of our algorithm is to explore T' top-down in the depth-first searching manner. To enable the operations that are conducted by *PathStack* each time when a node in q is chosen, we change the data structure as follows.

Instead of a list attached with each q_i in q, we associate v in T' with a list of nodes from q, denoted as $L(v)$, such that v satisfies the predicate of each node in the list. See Fig. 4(c) for illustration.

Based on such a data structure, the following algorithm can be easily implemented.

Algorithm *RefindPathStack*

1. *stack* ← root of *T'*;
2. **while** ¬ empty(*stack*) **do**
3. { v ← pop(*stack*);
4. **for** each q_i in q **do**
5. **while** (¬ empty($S(q_i)$) ∧ (*RightPos*(top($S(q_i)$)) < *LeftPos*(α(v)) **do** pop($S(q_i)$);
6. **for** each q_j in $L(v)$ **do**
7. {push($S(q_j)$, α(v));
8. establish *pointer to* top($S(parent(q_j))$);
9. **if** q_j is a leaf node
10. **then** {call *showSolutions*($S(q_j)$, 1); pop($S(q_j)$);
12. }
13. push all the children of v onto *stack*;
14. }

Example 3. When we apply our method to find all the paths in *T*, which match *q*, we will first generate the matching subtree *T'* of *T* and associate each *v* in *T* with a list as shown in Fig. 4(c). Then, we run *RefindPathStack* over *T'* and *q*. During this process, *T'* will be searched top-down as shown below.

step 1: v_1 is visited; and $\alpha(v_1)$ will be pushed onto $S(q_1)$ and $S(q_3)$, respectively, as shown in Fig. 5(a).

step 2: v_2 is visited; and $\alpha(v_2)$ will be pushed onto $S(q_2)$. Meanwhile, a pointer to the top of the stack of q_2's parent will be established as shown in Fig. 5(b).

step 3: v_4 is visited; and $\alpha(v_4)$ will be pushed onto $S(q_4)$. Also, a pointer to the top of the stack of q_4's parent will be established as shown in Fig. 5(c).

step 4: v_5 is visited. Since $RightPos(\text{top}(S(q_4))) = 4 < LeftPos(\alpha(v_5)) = 6$, v_4 is popped out from $S(q_4)$. After that, $\alpha(v_5)$) is pushed onto $S(q_1)$ and $S(q_3)$, respectively, as shown in Fig.5(d).

step 5: v_6 is visited. $\alpha(v_6)$ will be pushed onto $S(q_2)$, and a pointer as shown in Fig. 5(e) will be created.

step 6: v_7 is visited. and $\alpha(v_7)$ will be pushed onto $S(q_1)$ and $S(q_3)$, respectively. In addition, a pointer to its parent is generated as shown in Fig. 5(f).

step 7: v_8 is visited. $\alpha(v_8)$ will be pushed onto $S(q_4)$ and a pointer is established as shown in Fig. 5(g). □

During the execution of *RefindPathStack*, each *v* in *T'* is accessed only once. Moreover, each time when a *v* is visited, the top elements of *m* stacks are checked and at most *m* nodes are pushed onto the stacks. Let δ_{ij} denote the number of elements removed from $S(q_j)$ in the *i*-th iteration. Therefore, the total cost is bounded by

$$O\left(\sum_{i=1}^{n}\sum_{j=1}^{m}(1+\delta_{ij})\right)$$

$$= O\left(\sum_{i=1}^{n}\sum_{j=1}^{m}1\right) + \sum_{i=1}^{n}\sum_{j=1}^{m}\delta_{ij}$$

$$= O(m \cdot n).$$

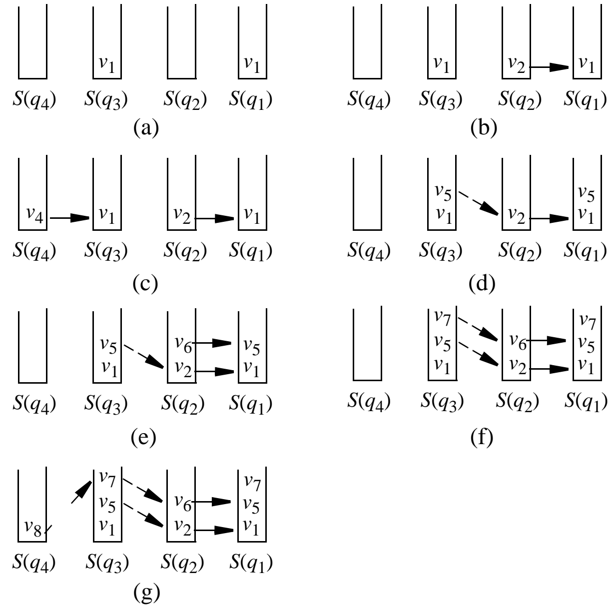

Fig. 5. Sample trace of *RefindPathStack*

3. Refined twigstack

From the previous section, we can see that the elimination of *PathStack*'s redundancy is relatively straightforward. But it is more challenging to remove redundancy from *TwigStack* [2], which is used to handle more complicated cases that the query is a non-trivial tree, but containing only *d*-edges. As in Section 2, we will first describe *TwigSatck* and analyze its time complexity in 3.1. Then, in 3.2, we give a new algorithm, which substantially improves *TwigStack*.

3.1 Description of TwigStack

Algorithm *TwigStack* operates in two phases. In the first phase, all paths matching individual query root-to-leaf paths are produced. In the second phase, these matching paths are merge-joined to create the answers to the query twig pattern.

In order to generate all the matching paths, *TwigStack* uses the same data structures as *PathStack*, but work in a quite different way.

Algorithm *TwigStack*(q)

(*phase 1*)

1. **while** ¬ *end*(q) **do**
2. { q_{act} ← *getNext*(q);
3. **if** (q_{act} is not the root) **then**

4. *cleanStack*(S(*parent*(q_{act})), *LeftPos*(*next*(L(q_{act})));
5. **if** (q_{act} is the root of q) $\vee \neg$ empty(S(*parent*(q_{act})))
6. **then**
7. {*cleanStack*(S(q_{act}), *LeftPos*(*next*(L(q_{act})));
8. *moveStreamToStack*(L(q_{act}), S(q_{act}), pointer to top(S(q_{act})));
9. **if** (q_{act} is a leaf node) **then**
10. {*showSolutions*(S(q_{act}), 1); pop(S(q_{act}));}
11. }
12. **else** *advance*(L(q_{act}));
13.}

(*phase 2*)

14. *mergeAllPathSolutions*();

Function *getNext*(q)

1. **if** (q is a leaf node) **then** return q;
2. let q_1, ..., q_k be the children of q;
3. **for** i = 1 to k **do**
4. {$n_i \leftarrow getNext(q_i)$;
5. **if** ($n_i \neq q_i$) **then** return n_i;}
6. $n_{min} \leftarrow min\{LeftPos(n_1), ..., LeftPos(n_k)\}$;
7. $n_{max} \leftarrow max\{LeftPos(n_1), ..., LeftPos(n_k)\}$;
8. **while** (*RightPos*(*next*(L(q)) < *LeftPos*(*next*(L(n_{max})) **do**
9. *advance*(L(q));
10. **if** (*LeftPos*(*next*(L(q)) < *LeftPos*(*next*(L(n_{min})) **then** return q;
11. **else** return n_{min};

Procedure *cleanStack*(S, *actL*)

1. **while** ($\neg$ empty(S) $\wedge$ (*RightPos*(top(S) < *actL*) **do**
2. pop(S);

The above algorithm is a more complicated process than *PathStack*. First, *getNext* is quite different from *getMinSource*, by which a q_i from q is chosen iff the following two conditions are satisfied:

(i) Let q_{i_1}, ..., q_{i_k} be the children of q_i. Let v be the next node in $L(q_i)$ to be processed. Then, v has a descendant u such that $\alpha(u)$ in $L(q_{i_j})$ for each q_{i_j} $(1 \leq j \leq k)$.

(ii) Each u recursively satisfies the first property.

In this way, each solution to each individual query root-to-leaf path is guaranteed to be merge-joinable with at least one solution to each of other root-to-leaf paths. Therefore, the dominate cost of the first phase is the execution time of *getNext*.

Let n_i be the size of $L(q_i)$. Then, the main **while**-loop in *TwigStack* will be iterated $\sum_{i=1}^{m} n_i$ times since the termination condition of this **while**-loop is when all the elements in all $L(q_{leaf})$'s are exhausted. In each iteration, the procedure *getNext* will be invoked and all the nodes in the query tree will be accessed. Let λ_{ijk} be the number of elements in $L(q_k)$ checked when node q_k is visited during the (i, j)-th execution of *getNext*. Then, the worst-case cost is bounded by

$$O\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}(1+\lambda_{ijk})\right)$$

$$= O\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}1\right) + O\left(\sum_{i=1}^{m}\sum_{j=1}^{n_i}\sum_{k=1}^{m}\lambda_{ijk}\right)$$

$$= O\left(\sum_{i=1}^{m} m \cdot n_i\right) + O(m \cdot n) = O(m^2 \cdot n).$$

3.2 Removing redundancy from TwigStack

Now we begin to discuss how the redundancy of *TwigStack* can be removed. As with *TwigStack*, we will associate each node q_i in q with a data stream $L(q_i)$ the following conditions:

(i) For each $v \in L(q_i)$, v matches the predicate at q_i.

(ii) Let q_{i_1}, ..., q_{i_k} be the children of q_i. v has a descendant v' matching q_{i_j} for $j \in \{1, ..., k\}$.

(iii) Each of the nodes v' recursively satisfies (ii).

Obviously, these three conditions correspond to the two properties given in the previous subsection, for any node going onto a stack. There is not anything new. However, not like *getNext* in *TwigStack*, which chooses nodes from q to handle and in fact each time finds a next v in T' to be put in some stack (by multiple executions), we generate all $L(q_i)$'s in one scan, which enables us to avoid a great number of repeated accesses to query nodes.

For this purpose, we maintain two $m \times n$ ($m = |q|$, $n = |T'|$) matrices.

1. The nodes in both q and T' are numbered in postorder, and the nodes are then referred to by their postorder numbers.
2. In the first matrix, each entry c_{ij} ($i \in \{1, ..., m\}$, $j \in \{1, ..., n\}$) has value 0 or 1. If $c_{ij} = 1$, it indicates that $i \in L(j)$ and for each child of i, j has a descendant satisfying the predicate at it. Otherwise, $c_{ij} = 0$. This matrix is denoted by $c(q, T')$.
3. In the second matrix, each entry d_{ij} ($i \in \{1, ..., m\}$, $j \in \{1, ..., n\}$) is defined as follows. If j has a descendant j' such that $c_{ij'} = 1$, then $d_{ij} = 1$; otherwise $d_{ij} = 0$. This matrix is denoted by $d(q, T')$.

The following algorithm can be used to generate the values for these two matrices.

Initially, $c_{ij} = 0$ and $d_{ij} = 0$ for all i and j. During the execution of the algorithm, the values of c_{ij}'s will be changed according to conditions (i) and (ii) described above; and d_{ij}'s will be changed to record whether a node j in T' has a descendant j' that matches a certain node i in q.

Algorithm *matrixGeneration*(T', q)

Input: tree T' (with nodes 1, ..., n) and tree q (with nodes 1, ..., m)
Output: $c(q, T')$ with values created.

```
begin
1.  for u := 1, ..., m do {
2.     for v := 1, ..., n do
3.       {if v satisfies the predicate at u then
4.         let u_1, ..., u_k be the children of u;
5.         if d_{u_1 v} ∧ ... ∧ d_{u_k v} = 1 then c_{uv} ← 1;
6.       }
7.   let v_1, v_2, ..., v_h be the nodes such that c_{u v_p} = 1 (1 ≤ p ≤ h);
8.   let {w_1, ..., w_r} be a set such that each node in it is an ancestor
     of some v_p (1 ≤ p ≤ h). Set d_{u w_l} = 1 for each w_l (1 ≤ l ≤ r).
9.  }
end
```

To see how the above algorithm works, we should first notice that both T' and q are both postorder-numbered. Therefore, the algorithm proceeds in a bottom-up way (see line 1 and 2). For any node u in q and any node v in T', if v satisfies the predicate at u, we will check each child u_i of u to see whether there exists a descendant of v that matches u_i (see line 5). If it is the case, c_{uv} will be set to 1.

In line 7 and 8, we change d_{ij}'s according to the newly changed c_{ij}'s.

Example 3. As an example, consider the trees shown in Fig. 6. The nodes in them are postorder numbered.

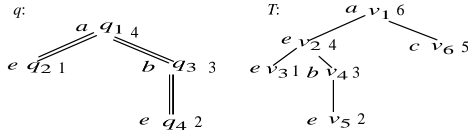

Fig. 6. Labeled trees and postorder numbering

When we apply the algorithm to these two trees, $c(q, T)$ and $d(q, T)$ will be created and changed in the way as illustrated in Fig. 7, in which each step corresponds to an execution of the outmost **for**-loop.

In step 1, we show the values in $c(q, T)$ and $d(q, T)$ after node 1 in q is checked against every node in T. Since node 1 in q matches node 1, 2 and 4 in T, c_{11}, c_{12}, and c_{14} are all set to 1. Meanwhile, for all those nodes that are an ancestor of 1, 2, or 4 in T, the corresponding entries in $d(q, T)$ will be changed. So we have all d_{11}, d_{12}, d_{13}, d_{14}, and d_{16} set to 1 (see line 7 and 8).

In step 2, the algorithm generates the matrix entries for node 2 in q, which is done in the same way as for node 1 in q.

In step 3, node 3 in q will be checked against every node in T, but matches only node 3 and 5 in T. Since it is an internal node, its children will be further checked. For node 3 in T, it is done by checking d_{23}, which is equal to 1. So node 3 in T matches node 3 in q. For node 5 in T, since d_{35} is 0, it does not match node 3 in q. In step 4, since node 6 in T matches node 4 in q and both d_{16} and d_{36} are equal to 1, c_{16} is set to 1 (d_{16} is then set to 1 by executing line 7 and 8). □

step 1:

$c(q, T)$: $\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ $d(q, T)$: $\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

step 2:

$c(q, T)$: $\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ $d(q, T)$: $\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

step 3:

$c(q, T)$: $\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ $d(q, T)$: $\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

step 4:

$c(q, T)$: $\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ $d(q, T)$: $\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

(Rows numbered 1–4, columns numbered 1–6.)

Fig. 7. Sample trace

Proposition 1. Algorithm *matrixGeneration*(T, q) computes the values in $c(q, T)$ and $d(q, T)$ correctly.

Proof. The proposition can be proved by induction on the sum of the heights of T and q. □

Proposition 2. Algorithm *matrixGeneration*(T, q) requires $O(n \cdot m)$ time and space, where $n = |T|$ and $m = |q|$.

Proof. During the whole process, against each node u in q, all the nodes v in T is checked and for each v all its children will be examined. Therefore, this part of time is bounded by

$$O\left(\sum_{u=1}^{m} \sum_{v=1}^{n} d_v\right) = O\left(\sum_{u=1}^{m} n\right) = O(n \cdot m),$$

where d_v represents the outdegree of node v in T.

In addition, after each u in q is checked, for all those nodes in T, which are an ancestor of some node that

matches u, the corresponding matrix entries in $d(q, T)$ will be established. But this operation needs only $O(n)$ time if we proceeds as follows. Each time when we search T bottom-up from a node v that matches u to find all its ancestors, we mark each node encountered and stop whenever we meet such a mark (made by a previous searching). So at most $O(n)$ nodes will be checked and the total time of this part of operations is bounded by $O(n \cdot m)$.

Obviously, to maintain $c(q, T)$ and $d(q, T)$, we need $O(n \cdot m)$ space. □

In terms of the matrix $c(q, T)$, it is an easy task to create $L(q_i)$ for each q_i in q as illustrated in Fig. 8(a).

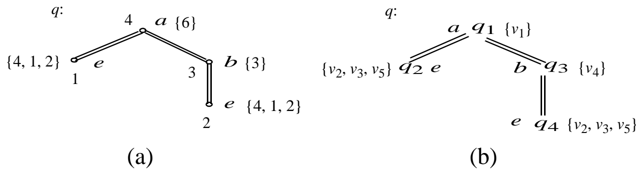

Fig. 8. Illustration for $L(q_i)$'s

Fig. 8(b) is the same as Fig. 8(a). But in this figure we use node names in $L(q_i)$ instead of their postorder numbers.

Concerning $L(q_i)$, we should pay attention to the following:

(1) The nodes (represented by their quadruple) in $L(q_i)$ are sorted by their (DocId, LeftPos) values (not according to their postorder numbers).

(2) Each node in $L(q_i)$ satisfies the condition (i) and (ii) given in 3.1.

Using such a data structure, the algorithm *TwigStack* can be substantially improved. As with *RefinedPathStack*, we use a stack to control the searching of q in the depth-first fashion. Each entry in the stack is a pair (q_i, v_j), where $q_i \in q$ and $v_j \in T'$.

Finally, we notice that *getNext*( ) is not used since all the values to be produced by executing *getNext*( ) are pre-calculated and incorporated into $L(q_i)$'s.

Algorithm *RefinedTwigStack*(q)

(*phase 1*)
1.Repeat the following until all $L(q_i)$ become empty;
2. {let *LeftPos*(q_i) be the least such that ¬ empty($L(q_i)$);
3. push(*stack*, (q_i, *next*($L(q_i)$))); *advance*($L(q_i)$);
4. **while** ¬ empty(*stack*) **do**
5. {(u, v) ← pop(*stack*);
6. **if** (u is not the root) **then**
7. *cleanStack*(S(*parent*(u)), LeftPos(v));
8. **if** (u is the root of q) ∨ ¬ empty(S(*parent*(u)))
9. **then**
10. {*cleanStack*($S(u)$, LeftPos(v));
11. push($S(u)$, v, pointer to top(S(*parent*(u)));
12. **if** (u is a leaf node) **then**
13. {*showSolutions*($S(u)$, 1); pop($S(u)$);}
14. }
15. **else** *advance*($L(u)$);
16. let q_1, ..., q_l be the children of u;
17. **for** $j = l$ to 1 **do**
18. {**while** *next*($L(q_j)$) is not a descendant of v **do** *adavance*($L(q_j)$);
19. push(stack, (q_j, *next*($L(q_j)$)));}
20. }}
(*phase 2*)
21. *mergeAllPathSolutions*();

Example 4. Continue with Example 3.

By using our method, we will first generate $L(q_i)$ for each q_i as shown in Fig. 8(b). Then, we will search the twig pattern q as follows.

step 1: At the very beginning, the node q_1 has the least LeftPos value and $L(q_1)$ is not empty. Push (q_1, v_1) into *stack*.

step 2: In the following **while**-loop, the whole query tree will be traversed.
When we meet q_2, which is a leaf node, we have a configuration as shown in Fig. 9, which contains the first matching path: $v_2 \rightarrow v_1$. By using *showSolution*( ), we can store it in a tuple.

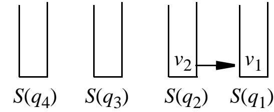

Fig. 9. The first matching path

step 3: When we meet q_3, which is an internal node, the stacks will be changed as shown in Fig. 10.

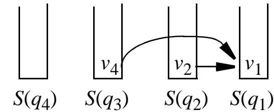

Fig. 10. Illustartion for stack changes

step 4: When we meet another leaf node q_4, we will get the second matching path: $v_5 \rightarrow v_4 \rightarrow v_1$, stored in stacks as shown in Fig. 11.

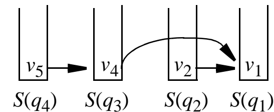

Fig. 11. The second matching path

step 5: Now *stack* (used to control the searching of q) is empty. We will try to find another node (in q) with the least LeftPos value and a non-empty list. It is q_2. In $L(q_2)$, we have two elements left: $\{v_3, v_5\}$. Push (q_2, v_3) into *stack*.

step 6: In the **while**-loop, q_2 is accessed and the stack configuration is changed as shown in Fig. 12, from which we can take the third matching path:

$v_3 \rightarrow v_1$.

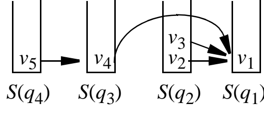

Fig. 12. The third matching path

step 7: Since the lists associated with all the other nodes are empty now, *stack* becomes empty once again. As in step 5, we will try to find a node (in q) with the least LeftPos value and a non-empty list. It is q_2 once again. For it, we have $L(q_2) = \{v_5\}$. Proceeding as above, the stacks will be changed as shown in Fig. 13. From this, the fourth matching path: $v_5 \rightarrow v_1$ can be obtained. □

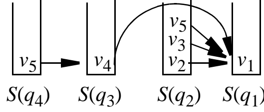

Fig. 13. The fourth matching path

The above algorithm works almost in the same way as *TwigStack*. The only difference is that in *TwigStack*, *getNext* is executed multiple times while in *RefinedTwigStack*, *getNext* is replaced with *matrixGeneration*, which is performed only once.

The time complexity of *RefinedTwigStack* is easy to analyze. In the whole process, each node v in a $L(q_i)$ is accessed only once. So the total cost is bounded by

$$O\left(\sum_{i=1}^{m} L(q_i)\right) = O(m \cdot n).$$

4. General cases

The method discussed in Section 3 can be easily extended to handle general cases that a query tree contains both c-edges and d-edges. For this purpose, we define a third matrix $p(q, T)$ as follows.

An entry $p_{ij} = 1$ indicates that there exists some child k of j, which 'matches' i, *i.e.*, $c_{ik} = 1$; otherwise, $p_{ij} = 0$.

Accordingly, the algorithm *matrixGeneration* should be slightly changed so that the manipulation of $p(q, T)$ is involved.

Algorithm *generalMatrixGeneration*(T, q)

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

Output: $c(q, T)$ with values created.

begin

1. **for** u := 1, ..., m **do** {
2. **for** v := 1, ..., n **do**
3. {**if** v satisfies the predicate at u **then**
4. let u_1, ..., u_k be the c-children of u;
5. let u_1', ..., u_g' be the d-children of u;
5. **if** $e_{u_1 v} \wedge ... \wedge e_{u_k v} = 1$ and $d_{u_1' v} \wedge ... \wedge d_{u_g' v} = 1$
6. **then** $c_{uv} \leftarrow 1$;
7. }
8. let v_1, v_2, ..., v_h be the nodes such that $c_{uv_p} = 1$ $(1 \leq p \leq h)$;
9. let $\{w_1, ..., w_r\}$ be a set such that each node in it is an ancestor of some v_p $(1 \leq p \leq h)$. Set $d_{uw_l} = 1$ for each w_l $(1 \leq l \leq r)$.
10. let $\{t_1, ..., t_s\}$ be a set such that each node in it is a parent of some v_p $(1 \leq p \leq h)$. Set $d_{ut_l} = 1$ for each t_l $(1 \leq l \leq s)$.
11. }

end

Since each node u in q may have both c- and d-children, each time when checking it against a node v in T we need to check the corresponding entries in both $d(q, T)$ and $p(q, T)$ (see line 5). In addition, besides the computation of new value for some entries in $d(q, T)$ in each step, we need also to compute new values for the corresponding entries in $p(q, T)$ (see line 10).

Example 4. Consider T and q shown in Fig. 14. Especially, q is a general query tree, containing both c- and d-edges.

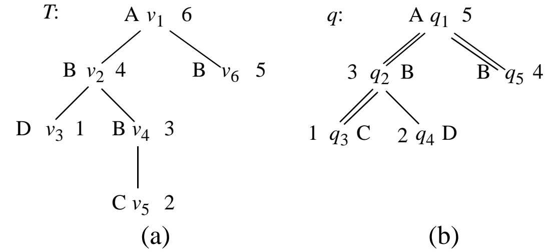

Fig. 14. A query tree containing c- and d-edges

Applying the above algorithm to T and q shown in Fig. 14, we will generate three matrices as shown in Fig. 15.

$c(q, T)$:

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	1	0	0	0	0	0
3	0	0	0	1	0	0
4	0	0	1	1	1	0
5	0	0	0	0	0	1

$d(q, T)$:

	1	2	3	4	5	6
1	0	1	1	1	0	1
2	1	0	0	1	0	1
3	0	0	0	1	0	1
4	0	0	1	1	1	1
5	0	0	0	0	0	1

$p(q, T)$:

	1	2	3	4	5	6
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	0

Fig. 15. Three matrices

Special attention should be paid to c_{34}. It is set to 1 since we have $d_{14} = 1$ and $p_{24} = 1$ before the calculation of this entry is performed.

The modification to *RefinedTwigStack* is quite trivial: just one line needs to be changed as below.

... ...

18. {**while** *next*($L(q_j)$) is not a descendant of v if q_j is a *d*-child or *next*($L(q_j)$) is not a child of v if q_j is a *c*-child **do**

adavance($L(q_j)$);

... ...

In this way, the *c*-edges can be correctly handled.

Finally, from the above discussion, we can also see that for any query tree containing both *c*- and *d*-edges, the time complexity remains $O(m \cdot n)$.

5. Conclusion

In this paper, a new method is discussed, which substantially improves the method proposed in [2] for doing twig joins that are identified as the core operation for query evaluation in XML databases. Concretely, our method improves the algorithm *PathStack* and *TwigStack* presented in [2] from $O(m^2 \cdot n)$ to $O(m \cdot n)$, where m and n are the sizes of the query tree and document tree, respectively.

References

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

[2] N. Bruno, N. Koudas, and D. Srivastava, Holistic Twig Hoins: Optimal XML Pattern Matching, in *Proc. SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 2002.

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

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

[5] A. Deutch, M. Fernandex, D. Florescu, A. Levy, D.Suciu. "A Query Language for XML," WWW'99.

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

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

[8] 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.

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

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

[11] 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.

[12] 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.

[13] World Wide Web Consortium. XML Path Language (XPath), W3C Recommendation, Version 1.0, November 1999. See http://www.w3.org/TR/xpath.

[14] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Recommendation, Version 1.0, Dec. 2001. See http://www.w3.org/TR/xquery.

[15] 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.