Tree Inclusion Algorithm, Signatures and Evaluation of Path-Oriented Queries

Yangjun Chen*, Yong Shi+, and Yibin Chen*
*Dept. of Applied Computer Science
University of Winnipeg, Manitoba, Canada R3B 2E9
+Dept. of Computer Science
University of Manitoba, Manitoba, Canada R3T 2N2

ABSTRACT

In this paper, a method to evaluate path-oriented queries in document databases is proposed. The main idea of this method is to handle the evaluation of a path-oriented query as a tree inclusion problem. A new algorithm for tree-inclusion is discussed, which integrates a top-down process into a bottom-up searching strategy. On the one hand, the algorithm can be arranged to access the data on disk page-wise and fits therefore within a database environment. On the other hand, the algorithm can be combined with the signature indexing technique to cut off useless subtree inclusion checkings as early as possible. Experiments have been conducted to compare this method with some existing approaches, which shows that the integration of the signatures into the top-down tree inclusion is highly promising.

Categories & Subject Decriptors: H.2.4
General Terms: Databases**,** Algorithms, Performance
Key Words: XML document, path oriented queries, tree inclusion, signatures, indexes

1. INTRODUCTION

In query languages proposed for XML, and even more generic SGML query languages, *path-oriented queries* play a prominent role. By "path-oriented" we mean queries that are based on the path expressions including element tags, attributes, and key words. As an example, consider the following query in XPath expression:

Q_1: hotel-room-reservation[name/text()]/location[city-or-ditrict [text() = 'Winnipeg']/address [//number[text() = '510']// street[text() = 'Portage Ave.'].

This path-oriented query asks for the name of the hotel located in 510 Portage Ave., Winnipeg, and can be represented as a tree shown in Fig. 1(a). In the query, '/' is the child operator which selects from immediate child nodes, and '//' is the descendant operator which selects from arbitrary descendant nodes.

As another example, the query below is just a single path:

Q_2: letter//body/*/para [text() contain 'visited']

where [text() contain 'visited'] is a predicate, enquiring whether element para's text() contains a word 'visited'. This query is pictorially shown in Fig. 1(b).

In the rest of the paper, we mainly discuss how such queries can be evaluated.

A lot of work has been done on this issue [4, 8, 11, 12]. However, all the methods proposed fail to recognize that the evaluation of a path-oriented query is in essence a tree-inclusion problem, and no effort has been made in them to analyze the nature of this problem.

SAC'06, April 23-27, 2006, Dijon, France.

In the following, we show some of existing strategies and analyze their computational complexities. The first one is based on *Indexing elements and words* [12], the second is based on *Indexing paths and words* [8], and the third is based on indexing tree structures [11]. All these methods are somehow related to our method to be discussed.

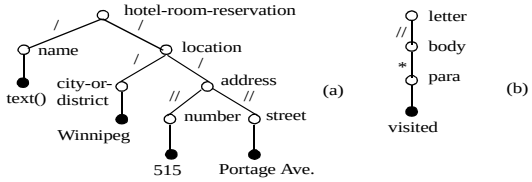


Fig. 1. Tree representation of XPath queries

Indexing elements and words

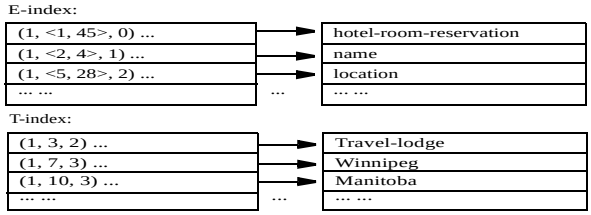
There is a lot of work that considers using relational database techniques to store and retrieve XML documents, such as those discussed in [4, 8, 11, 12]. Among them, the method discussed in [12] is inversion-based. In this method, two kinds of inverted indexes are established for text words and elements, by means of which a text word (or an element) is mapped to a list, which enumerates all the documents containing the word (or the element) and its positions within a document. To speed up a query evaluation, the position of a word (or an element) is recorded as follows:

- (*Dno*, *Wposition*, *level*) for a text word,
- (*Dno*, *Eposition*, *level*) for an element,

where *Dno* is its document number, *Wposition* is its position in the document, and *level* is its nesting depth within the document; *Eposition* is a pair: <*s*, *e*>, representing the positions of the start and end tags of an element, respectively. For instance, the document shown in Fig. 2(a) is indexed as illustrated in Fig. 2(b). The index for elements is called *E-index* and the index for words is called *T-index*.

```
<hotel-room-reservation filecod="1302">
    <name>Travel-lodeg</name>
    <location>
        <city-or-district>Winnipeg</city-or-district>
        <state>Manitoba</state>
        <country>Canada</country>
        <address>
            <number>500</number>
            <street>Portage Ave.</street>
            <post-code>R3B 2E9</post-code>
        </address>
    </location>
    <type>
        <rooms>one-bed-room</room>
        <price>$119.00</price>
    </type>
    <reservation-time>
        <from>April 20, 2002</from>
        <to>April 28, 2002</to>
    </reservation-time>
</hotel-room-reservation>
```

(a)



(b)

Fig. 2. A sample XML file and its inverted lists

The Author is supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Canada).

Let (d, x, l) be an index entry for an element a. Let (d', x', l') be an index entry for a word b. Then, a contains b iff $d = d$' and $x.s < x' < x.e$. Let (d'', x'', l'') be an index entry for another element c. Then, a contains c iff $x.s < x''.s$ and $x.e > x''.e$. Using these properties, some simple path-oriented queries can be evaluated. For instance, to process the query: /hotel-room-reservation/location/city-or-district [text() = Winnipeg], the inverted lists of 'hotel-room-reservation', 'location', 'city-or-district' and 'Winnipeg' will be retrieved and then their containment will be checked according to the above properties. In a relational database, E-index and T-index are mapped into the following two relations:

E-index(element, docno, begin, end, level), and

T-index(word, docno, wordPosition, level),

where the primary keys are underlined.

The index structures are efficient for simple cases, such as whether a word is contained in an element. However, in the case that a query is a non-trivial tree, the evaluation based on these index structures are an exponential time process. To see this, consider the query: /hotel-room-reservation/location/address/street [text() = Portage Ave.]. To evaluate this query, four joins have to be performed. They are the self-joins on E-index relation to connect 'hotel-room-reservation' and 'location', 'location' and 'address', and 'address' and 'street', as well as the join between E-index and T-index relations to connect 'street' and 'Portage Ave.' In general, for a document tree with n nodes and a query tree with m nodes, the checking of containment needs $O(n^m)$ time in the worst case.

Indexing paths and words

The above method is improved by Seo *et al*. [8] by introducing indexes on paths to reduce the number of joins as well as the sizes of relations involved in a join operation. This is achieved by establishing four relations to accommodate the inverted lists:

Path(pathID, path),

PathIndex(pathID, docno, begin, end),

Word(wordID, word), and

WordIndex(wordID, docno, pathID, position).

In this way, the number of joins is dramatically decreased. For example, to process the same query: /hotel-room-reservation/location/address/street [text() = Portage Ave.], only two joins are needed. The first join is between the Path and WordIndex relations with the join condition:

Path.path = 'hotel-room-reservation/location/address/ street' and

Path.pathID = WordIndex.pathID.

The second join is between the result R of the first join and the Word relation with the join condition:

R.wordID = Word.wordID and Word.word = 'Portage Ave.'.

In general, the query evaluation based on such an index structure needs k joins, where k is the number of the paths in a query. However, such a time improvement is at cost of memory space since in the relation Path the element names are repeatedly stored. Concretely, for a document with n nodes, the size of Path-relation is on the order of $O(n \cdot h)$, where h is the average height of a document tree. Therefore, the time complexity of this method is $O(k \cdot l \cdot n \cdot h)$, where l stands for the number of the (key) words in a document.

- indexing tree structures

In [11], a different method was proposed, which stores a document as a sequence: (a_1, p_1), ..., (a_i, p_i), ..., (a_n, p_n), where each a_i is an element or a word in the document, and p_i a path from the root to it. In the sequence, the pairs appear in the preorder of the document tree. For instance, the XML document shown in Fig. 2(a) will be stored in the following form:

1 (hotel-room-reservation,	∅)
2 (name,	hotel-room-reservation)
3 (Travel-lodge,	hotel-room-reservation.name)
4 (location,	hotel-room-reservation)
5 (city-or-district,	hotel-room-reservation.location)
6 (Winnipeg,	hotel-room-reservation.location.city-or-district)
... ...	... ...

In practice, all the paths can be stored in a separate set without repeated paths and in a pair such as (a_i, p_i), p_i is just an address. In this way, the space overhead can be reduced. However, if all the paths are different, the document storage can be huge and takes $O(n \cdot h)$ space as the method discussed in [8]. To speed up the query evaluation, each node is associated with a pair of integers in the same way as the method discussed in [12] to recognize the ancestorship of elements and words. Over such pairs, a B^+-tree is established. In addition, a trie structure (called suffix tree in [37]) is constructed over all the document sequences and labeled in the same way for a single document tree. Again, over such labels, another B^+-tree is built. When a query arrives, it will be transformed into a pair sequence which is scanned against all the document sequences. During this process, the labels associated with each pair in a document sequence are used to control the searching so that only those pairs in a document sequence are visited, which are descendants of the current pair. Especially, both the B^+-trees are employed to do the searching quickly. Since for each current pair, all its descendants will be checked and for checking each pair a path has to be scanned, the time complexity is $O(n \cdot m \cdot h)$. This time complexity is still very high, considering the state-of-the-art of the research on the so-called ordered tree inclusion problem, which can be adapted to handle path-oriented queries. Up to now, the best algorithm for solving this problem is $O(n \cdot p)$, where p represents the number of the leaf nodes of a query tree [2]. The method to be discussed below needs $O(n \cdot h_q)$ time, where h_q represents the height of a query tree.

The rest of the paper is organized as follows. In Section 2, we first describe the main ideas of our method and then discuss a new tree inclusion algorithm for the evaluation of path-oriented queries. In Section 3, we show how the signature technique can be integrated into it to cut off non-relevant branches. Finally, a short conclusion is set forth in Section 4.

2. QUERY EVALUATION BASED ON TREE INCLUSION AND SIGNATURES

Now we begin to discuss our method based on the tree inclusion. First, we give a general description of our method in 2.1. Then, we show why the evaluation of a path oriented query is a tree inclusion problem in 2.2. Next, we present a new algorithm, which is particularly suitable in a database environment in 2.3.

2.1 General description

Our method can be described as a pair <S, A>, where $S = s_1, ..., s_i, ..., s_n$ is the preorder sequence of a document tree and A represents a tree inclusion algorithm. In S, each s_i is quadruple (a_i, c_i, b_i, *signature*$_i$), where a_i is an element or a text, c_i is a pointer to the first child node of a_i, b_i is a pointer to the right sibling of a_i, and *signature*$_i$ is a bit string of a certain length. We use c_i and b_i to keep the tree structure, and *signature*$_i$ to index the subtree rooted at a_i. For example, the XML document shown in Fig. 2(a) can be stored in a way shown in Fig. 3.

1 (hotel-room-reservation,	2, ∅, 1001 00111110)
2 (name,	3, ∅, 1001 00011100)
3 (Travel-lodge,	∅, ∅, 1000 00110100)
4 (location,	5, 2, 1001 00111110)
5 (city-or-district,	6, ∅, 1001 00011100)
6 (Winnipeg,	∅, ∅, 1001 00010100)
... ...	... ...

Fig. 3. A document sequence

How to construct the signature associated with a node is discussed in 3.1.

When a query arrives, it will be transformed to a preorder sequence of the query tree: $Q = q_1, ..., q_j, ..., q_m$, where each q_j is a tuple of five elements (qa_j, qc_j, qb_j, *q-signature*$_j$, *q-label*$_j$). Here, *q-signature*$_j$ is generated in the same way as for the document tree nodes and *q-label*$_j$ is '/', '//', or '*', which is the label marking the edge from the parent of q_j to q_j. For example, The query tree shown in Fig. 1(a) will be transformed into a sequence shown in Fig. 4.

1 (hotel-room-reservation,	2, ∅, 1001 0011 1110, ∅)
2 (name,	3, 4, 1001 0001 1100, '/')
3 (text(),	∅, ∅, ∅, '/')
4 (location,	5, ∅, 1001 0011 1110, '/')
5 (city-or-district,	6, ∅, 1001 0001 1100, '/')
6 (Winnipeg,	∅, ∅, 1001 0001 0100, '/')
7 (address,	8, ∅, 1001 0000 1100, '/')
8 (number,	9, 10, 1001 0001 0100, '//')
9 (515,	∅, ∅, 1000 0001 0100, '//')
... ...	... ...

Fig. 4. A query sequence

After the query transformation, the tree inclusion algorithm *A* will be invoked to check all the document sequences against the query sequence. During this process, the signatures are used to cut off non-relevant documents, as well as useless subtrees within a document as early as possible.

2.2 Tree inclusion v.s. path oriented queries

As pointed out by Kilpelainen and Mannila[5], both documents and queries can be considered as *labeled trees* and the evaluation of a query can be thought of as a *tree embedding* (or say, *tree inclusion*) problem. In the following, we first define the concept of tree embedding. Then, we show that to evaluate a query, we will check whether the query tree is included in some document trees.

Definition 1 (*labeled tree*) A tree is called a labeled tree if a function *label* from the nodes of the tree to some alphabet is given, or each node in the tree is labeled. □

Obviously, an XML document can be represented as a tree with the internal nodes labeled with tags and the leaves labeled with texts; and a query as shown in Section 1 can also be represented as a labeled tree.

Definition 2 (*tree embedding*) Let *T* and *P* be two ordered, labeled trees. A mapping *M* from the nodes of *P* to the nodes of *T* is an embedding of *P* into *T* if it preserves labels, left-to-right ordering and ancestorship. That is, for all nodes *u* and *v* of *P*, we require that

a) $M(u) = M(v)$ iff $u = v$,
b) $label(u) = label(M(u))$,
c) *u* is an ancestor of *v* in *P* if and only if $M(u)$ is an ancestor of $M(v)$ in *T*, and
d) *v* is to the left of *u* iff $M(v)$ is to the left of $M(u)$. □

An embedding is *root preserving* if $M(\text{root}(P)) = \text{root}(T)$. According to [5], restricting to root-preserving embedding does not lose generality.

Example 1. As an example, consider the trees: *T* and *P* shown in Fig. 5(a), representing the document shown in Fig. 2(a) and the query shown in Fig.1(a), respectively. If a mapping as shown in Fig. 5(b) can be determined, we will have a tree-embedding of the query tree *P* into the document tree *T*, as indicated by the dashed lines.

For the query evaluation purpose, we'll return that document as one of the answers. □

However, the above example is just a special case of XPaths. To develop a general strategy, any algorithm to solve the tree inclusion problem has to be extended in such a way that the following issues can be addressed.

1. In an XPath query, different predicates may appear, such as =, <, >, 'contains', and so on.
2. The ordering of siblings in a query tree may be different from that in a document tree.
3. The edges in a query tree may be labeled with different connectors ('/' or '//'), or a wild card '*'.

The first problem can be handled by changing the label equivalence, i.e., $label(u) = label(M(u))$, to the corresponding predicate if *u* is a leaf node.

In order to tackle the second issue, we will change the sibling order in a query according to DTD if it is available. If the corresponding DTD does not exist, we simply use the lexicographical order of the names of the elements/attributes. In the case that a branch has multiple identical child nodes, the tree isomorphism problem cannot be avoided by enforcing sibling order. For example, a query of the form: /X[Y/Z/B]/Y/A can be represented as a tree shown in Fig. 6(a) or a tree shown in Fig. 6(b).

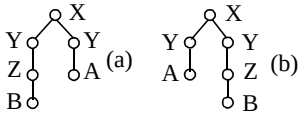


Fig. 6. Query trees

In this case, a sibling order cannot be specified lexicographically or by DTD schema. In order to find all matches, we have to check these two trees separately and unify their results.

The treatment of different connectors needs to change the algorithm itself. In the following, we will first present a new top-down algorithm and then discuss how to adapt it to different connectors.

2.3 A new top-down algorithm

A lot of methods have been developed to check tree inclusion in the theory research community, such as the methods discussed in [1, 2, 5, 6]. However, all these methods work in a bottom-up way and are not suitable for database applications since they all assume that both the target and pattern trees can be completely accommodated in main memory. In the case of large volume of data, it is not possible. For this reason, we design a new algorithm for the tree inclusion with a top-down process integrated into a bottom-up searching. This algorithm has the following advantages.

1. The time complexity of the algorithm is comparable to the best pure bottom-up method [2]. In fact, the method in [2] needs $O(n \cdot P_l)$ time while ours needs only $O(n \cdot P_h)$ time, where P_l stands for the number of the leaf nodes in *P* and P_h for the height of *P*.
2. Our algorithm works well in a database environment for the reason that it checks a target tree in a top-down fashion and each time only part of the target tree is manipulated.
3. The top-down searching enables us to differentiate the edges labeled with '/','//', or '*' in a query tree; and handle them in different ways.

- Algorithm description

In the following discussion, $T = <t; T_1, ..., T_k>$ represents a tree, where $t = root(T)$ and $T_1, ..., T_k$ are the subtrees of *t*; and $G = <P_1, ..., P_q>$ represents a forest containing subtrees $P_1, ..., P_q$.

Our algorithm attempts to find the number of subtrees j (≥ 0) within an ordered forest $G = <P_1, ..., P_q>$ ($q \geq 1$), which are able to be included in a target tree *T*. If $j = q$, we say that *G* is included in *T*. If $j < q$, then only the subtrees $P_1, ...,$ and P_j can be included in *T*. Let $p_1, ..., p_q$ be the roots of $P_1, ..., P_q$, respectively; and *t* be the root of *T*. Since a forest *G* does not have a root, we create a virtual node p_v to serve as a substitute for $root(G)$. Thus, $root(G)$ will return p_v if $G = <P_1, ..., P_q>$ with $q > 1$ (i.e., *G* is a real forest), and will return p_1 if $q = 1$.

There are three cases that need to be considered when designing an algorithm to check the tree embedding:

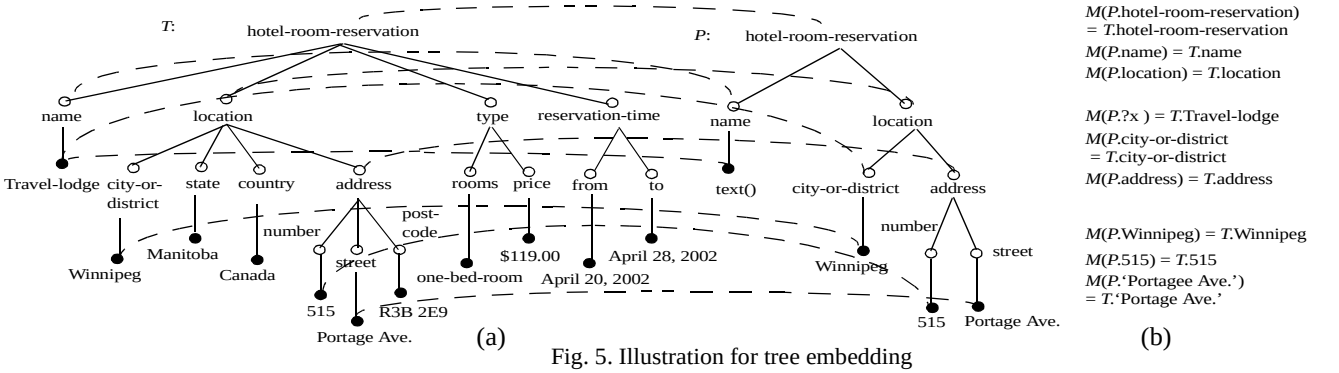


Fig. 5. Illustration for tree embedding

Case 1: *root*(G) ≠ p_v (*i.e.*, G = <P> is a tree and *root*(G) = p), and label(p) ≠ label(t). If G is included in T, then there must exist a subtree T_i of t such that it contains the whole G. The algorithm should return 1 if an embedding can be found and 0 if it cannot. (See Fig. 7 for illustration.)

For instance, if the root t of the document tree shown in Fig. 5 does not match the root p of the query tree, we will try to find a subtree of t, which includes the whole query tree.

Case 2: *root*(G) ≠ p_v (*i.e.*, G = <P> and *root*(G) = p), and label(p) = label(t). Let <P_1, ..., P_l> ($l \geq 0$) be the forest of subtrees of p and <T_1, ..., T_k> the forest of subtrees of t. If G is included in T, there must exist two sequences of integers: k_1, ..., k_g and l_1, ..., l_g ($g \leq l$) such that T_{k_i} includes <$P_{l_{i-1}+1}$, ..., P_{l_i}> (i = 1, ..., g, l_0 = 0, l_g = l), where <$P_{l_{i-1}+1}$, ..., P_{l_i}> represents a forest containing subtrees $P_{l_{i-1}+1}$, ..., and P_{l_i}. Thus, if l_g = l, the algorithm should return 1 since we have a root preserving inclusion of G in T. Otherwise, it should return 0. (See Fig. 8 for illustration.)

For instance, since the root t of the document tree shown in Fig. 5 matches the root p of the query tree, we will first try to find all those subtrees of p (the root of the query tree), which can be included in the first subtree of t. Then, we will try to find a second group of subtrees of p, which can be included in the second subtree of t, and so on.

Case 3: *root*(G) = p_v (*i.e.*, G is a forest) and there exists an integer j ($0 \leq j \leq q$) such that <P_1, ..., P_j> is included in T. If j = q, then the whole G is able to be included in T. There are two possibilities to be considered when looking for j. The first possibility is similar to Case 2, where there are two sequences of integers: k_1, ..., k_g and l_1, ..., l_g ($g \leq q$), which represent the order, in which the subtrees of *root*(G) are included in the subtrees of *root*(T). In this case, j = l_g. If j = 0, we will check the second possibility to see whether there exists a root preserving inclusion of P_1 in T, i.e., label(p_1) = label(t) and the subtrees of p_1 are included in the subtrees of t. In this case, j = 1. (See Fig. 9 for illustration.)

For instance, if we want to check whether the document tree shown in Fig. 5 includes the two query trees shown in Fig. 1, we will first try to find whether these two trees can be covered by the subtrees of the root of the document tree (possibility 1). Since it is not the case, we will check whether the document tree contains the first one of the two query trees (possibility 2).

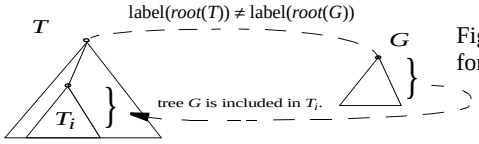


Fig. 7. Illustration for Case 1

In terms of above analysis, we give our algorithm which consists of two functions: *top-down-process*(T, G) and *bottom-up-process*(T, G). By *top-down-process*(T, G), the first input is a tree and the second can be a tree or a forest. By *bottom-up-process*(T, G), the first input is a tree and the second is a forest. In the algorithm, a forest is always handled as a tree with a virtual root.

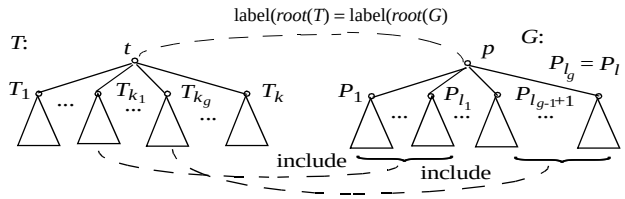


Fig. 8. Illustration for Case 2

Functionally, *top-down-process*( ) is designed to handle Case 1, Case 2, and the second possibility in Case 3 while *bottom-up-process*( ) is only for the first possibility in Case 3.

function *top-down-process*(T, G)
input: T = <t; T_1, ..., T_k>, G = <p; P_1, ..., P_q>
(*p may or may not be a virtual node.*)
output: if *root*(G) is virtual, returns $j \geq 0$; else returns 1 if T includes G; otherwise returns 0.

```
begin
1.   if root(G) is virtual then
2.     {if (|T| < |P1| + |P2| or p has only one child)
3.       then G := P1;
4.       else {j := bottom-up-process(T, G);
5.             if (j = 0 and label(t) = label(P1's root))
               (*second possibility in Case 3*)
6.             then {change P1's root to a virtual node;
                     x := bottom-up-process(T, P1);
7.                   if (x = the number of the children of P1's root)
                     then j := 1 else j := 0;}
8.                 return j;}}
9.     if |T| < |G | return 0;
10.    else {    if (label(t) = label(p))                (*handling Case 2*)
11.            then {p := virtual node;
12.                  j := bottom-up-process(T, G);
13.                  if (j = l) then return 1 else 0;}
14.           else {if t is a leaf then return 0;(*handling Case 1*)
15.                 i := 1;
16.                 while (i ≤ k) do
17.                 {x := top-down-process(Ti, G);
18.                  if x > 0 then return 1;
19.                  i := i + 1;}
20.                 return 0;} }
end
```

function *bottom-up-process*(T, G)
input: T = <t; T_1, ..., T_k>, G = <p; P_1, ..., P_q>
output: j - an integer

```
begin
1.     j := 0; i := 1;
2.     while (j < q and i ≤ k) do
3.       {  x := top-down-process(Ti, G);
4.          j := j + x; G := <p; Pj+1, ..., Pq>; i := i + 1;}
end
```

The algorithm begins from a checking to see whether G is a tree or a forest (see line 1 in *top-down-process*(T, G)). If it is a tree, the controls goes to line 9 to check whether $|T| < |G|$. If it is the case, return 0 to show that T does not include G. Otherwise, we will check whether the root t of T matches the root p of G (see line 10; it is Case 2). If it is the case, we will check whether the subtrees of p can be included in the subtrees of t by replacing p with a virtual node and then calling *bottom-up-process*(T, G) (see lines 11 - 12; in this case, we will be handling the second possibility of Case 3). Otherwise, we have case 1 and the control goes to line 14. If t is a leaf node, it is not possible for T includes G. So the algorithm returns 0. If t is not a leaf node, we will try to find whether there exists some subtree of t, which contains the whole G (see lines 15 - 20). If G is a forest (*i.e.*, its root p is a virtual node), we will check whether $|T| < |P_1| + |P_2|$ or p has only one child (see line 2). If $|T| < |P_1| + |P_2|$, T may includes P_1, but not P_1 and any other subtree together. So we will check wether T includes P_1. If p has only one child, it indicates that p has only one subtree left unchecked. So we will check wether T includes this subtree. In both cases, the control goes to line 9 and we will be handling Case 1 or Case 2. Otherwise, we will call *bottom-up-process*(T, G) to check the first possibility of Case 3 (see line 4). If the subtree of t cannot cover any subtree of p (*i.e.*, j = 0; see line 5), we will check the second possibility of Case 3 (see lines 5 - 8). We notice that in line 5, we check both j = 0 and label(t) = label(P_1's root). If t does not match P_1's root, the checking of the second possibility of Case 3 cannot be successful since in this case we have to find a subtree of t, which contains P_1. We have already done that and get j = 0, showing that such a subtree does not exist.

In *bottom-up-process*(T, G), we check the subtrees of t one by one against G_1, G_2, ..., G_j for some j, where each G_i ($1 < i \leq j$) is a forest obtained by removing those subtrees from G_{i-1}, which are found to be covered by the checked subtrees of t in the previous iteration steps (see lines 3 - 4.)

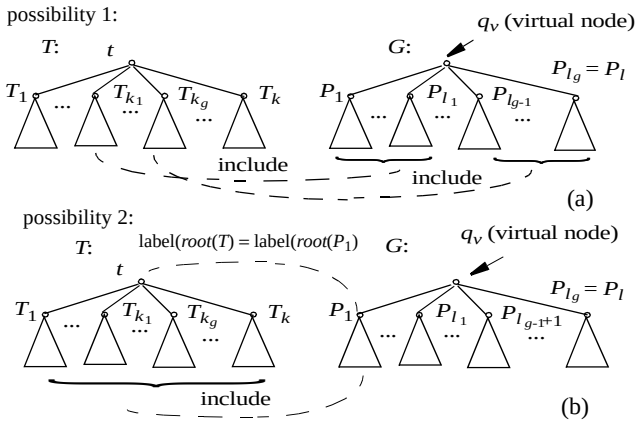


Fig. 9. Illustration for Case 3

- About handling '/', '//' and ''*

The above algorithm assumes that all the nodes on a path in a query tree are connected by '//'. To handle '/' and '*' correctly, we mark the edges in a query tree using '//', '/' and '*' according to the original query and deal with '/' or '*' as special constraints, which can be done easily as below. Each time when we call a top-down recursive call *top-down-process*(T_i, G) (see line 17 in Algorithm *top-down-process*(T, G) and line 3 in Algorithm *bottom-up-process*(T, G)), we will check the edge going to the root of G to see whether it is marked with '/', '//', or '*' and do the following:

- If the edge is marked with '/', check the root of T_i against the root of G immediately. If they don't match, *top-down-process*(T_i, G) will not be performed; but set x to 0. (x is the variable receiving the return value of *top-down-process*(T_i, G).) Otherwise, *top-down-process*(T_i, G) is conducted.
- If the edge is marked with '//', *top-down-process*(T_i, G) is directly conducted.
- If the edge is marked with '*', add a virtual node p_v as the parent of G's root and mark the edge from p_v to G's root with '/'. Denote this modified tree G' and call *top-down-process*(T_i, G').

3. Integrating signatures into tree inclusion

An advantage of the top-down process (integrated into a bottom-up strategy) is that we can integrate the signature technique into a tree inclusion to speed up the evaluation of a kind of XPath queries, i.e., the queries contain only 'contains' predicates. We assign each node v in T a bit string s_v (called a signature), and each node u in P a bit string s_u in such a way that if s_u matches s_v then the subtree T_v rooted at v may includes the subtree P_u rooted at u. Otherwise, T_v definitely does not contain P_u. Here, by "matching", we mean that for each bit set to 1 in s_u, the corresponding bit in s_v is also set to 1 while for a bit set to 0 in s_u, the corresponding bit in s_v can be 0 or 1. This method is similar to [7]. However, the combination of signatures into tree inclusion is the first effort to handle this problem.

3.1 Signature technique

The signature technique was originally introduced as a text indexing methodology. Nowadays, however, it is utilized in a wide range of applications, such as in office filing, hypertext systems, relational and object-oriented databases, as well as in data mining.

The main idea of the signature technique is that each word is processed separately by a hashing function. The scheme sets a constant number (m) of 1s in a [1..F] range. The resulting binary pattern is called the word signature. Each text is seen to be composed of fixed size logical blocks and each block involves a constant number (D) of non-common, distinct words. The D word signatures of a block are superimposed (bit OR-ed) to produce a single F-bit pattern, which is the block signature stored as an entry in the signature file [3]. To determine the length of signatures, we use the following formula [3]:

$$F \times \ln 2 = m \times D. \quad (1)$$

Fig. 10 depicts the signature generation and comparison process of a block containing three words (then D = 3), say "SGML", "database", and "information". Each signature is of length F = 12, in which m = 4 bits are set to 1. When a query arrives, the block signatures in the corresponding signature file are scanned and many non-qualifying blocks are discarded. The rest are either checked (so that the "false drops" are discarded; see below) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for word signatures. The query signature is then compared to every block signature in the signature file. Three possible outcomes of the comparison are exemplified in Fig. 10: (1) the block matches the query; that is, for every bit set in s_q, the corresponding bit in the block signature s is also set (*i.e.*, $s \wedge s_q = s_q$) and the block contains really the query word; (2) the block doesn't match the query (*i.e.*, $s \wedge s_q \neq s_q$); and (3) the signature comparison indicates a match but the block in fact doesn't match the search criteria (false drop). In order to eliminate false drops, the block must be examined after the block signature signifies a successful match.

block: ... SGML ... databases ... information ...

word signature:		
SGML		010 000 100 110
database		100 010 010 100
information	∨	010 100 011 000
object signature (OS)		110 110 111 110

queries:	query signatures:	matching results:
SGML	010 000 100 110	match with OS
XML	011 000 100 100	no match with OS
informatik	110 100 100 000	false drop

Fig. 10. Signature generation and comparison

For the tree inclusion problem, we can assign each label a signature in the same way as discussed above by using the method described in [3]. But the signature associated with a node in a document tree is defined as follows.

Definition 3. (*node signature*) Let v be a node in a tree T. If v is a leaf node, its signature s_v is equal to the signature assigned to its label. Otherwise, $s_v = s \vee S_{v_1} \vee ... \vee S_{v_n}$, where s represents the signature for the label associated with v, and S_{v_1}, ... , and S_{v_n} are the signatures of v's children: v_1, ..., v_n, respectively. □

Example 2. Consider the tree shown in Fig. 11(a). If the signatures assigned to the labels are those shown in Fig. 11(b). Each node in the tree will have a signature as shown in Fig. 11(c). □

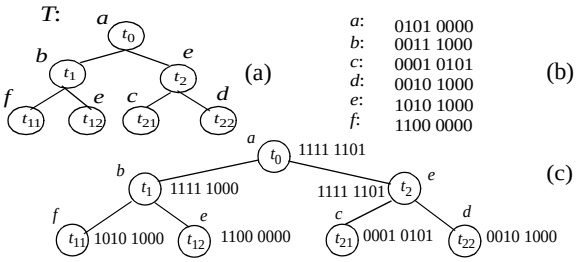


Fig. 11. Node signatures

Each time when we check a node u in P against a node v in T, we will first check their signatures. If they don't match, the subtree rooted at v will be cut off and not be searched any more, reducing the time overhead greatly.

Here, an important problem is how to determine the length of signatures. Due to the superimposing of signatures along the tree paths, the equation (1) shown above is not useful any more since it was established only for the simple structure of sequential signature files. However, if the length of signatures is not properly determined, as in an S-tree [9], the signatures near the root will be very heavy (*i.e*, populated with too many 1s in a signature) and the selectivity will be reduced dramatically. For this reason, we make the following analysis and develop a new method to estimate the signature length in

such a way that the above problem can be removed.

Consider two signatures s_1 and s_2. Assume that both of them are of length F and with m_1 and m_2 bits set to 1, respectively. Now, let $s = s_1 \vee s_2$. Obviously, s will possibly contain more 1s. To keep the ratio of 1s in s not increased, s should be set longer. The question is: how long should s be? Let l be the number of 1s in s and denote $\delta = l - m'$, where $m' = max(m_1, m_2)$. Then, $F + c\delta$ should be a reasonable length for s, where c is a constant and should be tuned for different applications. The value of δ can be estimated as follows.

Let λ be a random variable representing the number of positions, in which both s_1 and s_2 have 1s. Then, the *mathematical expectation* of λ can be calculated as below:

$$E\lambda = 1 \cdot p(\lambda = 1) + 2 \cdot p(\lambda = 2) + ... + m'' \cdot p(\lambda = m'') \quad (2)$$

where $m'' = min(m_1, m_2)$ and $p(\lambda = i)$ represents the probability that λ is equal to i. To calculate this probability, we use the following formula:

$$p(\lambda = i) = \frac{\binom{F-i}{m_1-i} \cdot \binom{F-m_1}{m_2-i}}{\binom{F}{m_1} \cdot \binom{F}{m_2}} \quad (3)$$

It is because the number of all the possible cases that s_1 has m_1 bits set to 1 and s_2 has m_2 bits set to 1 is $\binom{F}{m_1} \cdot \binom{F}{m_2}$; and the number of the cases that s_1 and s_2 have some bits set to 1 at the same i positions is $\binom{F-i}{m_1-i} \cdot \binom{F-m_1}{m_2-i}$.

Note that $l = m_1 + m_2 - \lambda$. Then, we have $\delta = l - m = m_1 + m_2 - \lambda - max(m_1, m_2)$.

Using the above formulas, we can determine the length of signatures for a tree as follows. First, we calculate the average number of key words in all the leaf nodes, which is used as the value of D to determine the initial values of F and m using formula (1). The key words can be identified by first using Connexor-analyzer [13] to figure out nouns and phrases and then using the method discussed in Chapter 3 in [10] to determine key words. Then, we compute the lengths of signatures for the internal nodes in a bottom-up way. That is, we first calculate the lengths for all those nodes, each of which is a parent of some leaf nodes. Then, we compute the lengths for the nodes at a higher lever. This process is repeated until the length of the signature for the root is computed, which will be used as the length of all the signatures to be generated.

3.2 Cutting off subtrees using signatures

Given two labeled trees T and P, we assign the signatures to their nodes in the same way. During the checking whether T includes P, we can use signatures to cut off some subtrees of T, which cannot contain the corresponding subtrees in P. This can be done by introducing the signature checkings into the algorithm *top-down-process*( ).

The following algorithm is almost the same as the algorithm *top-down-process*( ); but each time when we check whether a subtree in T includes a subtree in P, the corresponding signatures will be first checked. Of course, before the execution of the algorithm, the node signatures have to be first established for both T and P.

Algorithm *signature-tree-inclusion*(T, P)
Input: T, P
Output: if *root*(P) is virtual, return $j \geq 0$; else return 1 if T includes P; otherwise return 0.
begin
... ...
(*lines 1 - 8 are the same as in Algorithm *top-down-process*( ).*)
9. **if** $|T| < |P|$ return 0;
10. **else** {**if** p's signature *not match* t's signature **return** 0;
if (label(t) = label(p)
... ...
(*the rest part of the algorithm is the same as lines 11 - 20 in Algorithm *top-down-process*( ).*
end

We pay attention to line 10. Before we compare label(t) and label(p), we will first check their signatures. If P is a forest, a virtual root for P will be constructed and it does not have a signature. However, its signature can be easily established by superimposing the signatures of all its child nodes.

Example 3. Consider the trees T and P shown in Fig. 12. To check whether T includes P, we will assign signatures to the labels and the nodes in T and P in the same way. Assume that the assignment of the signatures to the labels is the same as shown in Fig. 11(b). Then, the checking of the subtree rooted at t_1 (in T) against the forest containing p_1 and p_2 (in P) can be avoided. It is because the signature for the virtual node of the forest (equal to 0011 1101) does not match the signature for t_1 (equal to 1111 1000). (See Fig. 12 for illustration.)

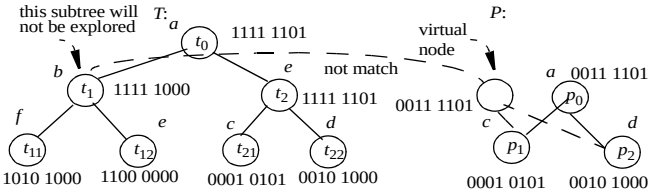


Fig. 12. Cutting subtrees using signatures

□

4. CONCLUSION

In this paper, a new strategy for evaluating path-oriented queries are discussed. The main idea of the query evaluation is a new algorithm for checking the inclusion of a query tree P in a document tree T, by which a top-down process is interleaved with a bottom-up computation. The algorithm has a comparable time complexity as the best bottom-up method, but needs no extra space. In addition, it is more suitable for a database environment and can be combined with the signature technique to get rid of useless checkings for subtree inclusion. In comparison with the methods based on the inverted indexes, our approach is more general and has a much better time complexity.

REFERENCES

[1] L. Alonso and R. Schott. On the tree inclusion problem. In *Proceedings of Mathematical Foundations of Computer Scienc*e, pages 211-221, 1993.

[2] W. Chen. More efficient algorithm for ordered tree inclusion. *Journal of Algorithm*s, 26:370-385, 1998.

[3] S. Christodoulakis and C. Faloutsos, "Design consideration for a message file server," *IEEE Trans. Software Engineering*, 10(2) (1984) 201-210.

[4] D. Florescu and D. Kossman, Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[5] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal of Computing*, 24:340-356, 1995.

[6] T. Richter. A new algorithm for the ordered tree inclusion problem. In *Proceedings* of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM), in *Lecture Notes of Computer Science (LNCS), volume 126*4, pages 150-166. Springer, 1997.

[7] Sangwon Park, Hyoung-Joo Kim: A New Query Processing Technique for XML Based on Signature. *DASFAA* 2001: 22-27.

[8] 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.

[9] E. Tousidou, P. Bozanis, Y. Manolopoulos, "Signature-based structures for objects with set-values attributes," *Information Systems*, 27(2):93-121, 2002.

[10] G. Salton and M.J. McGill, "*Introduction to Modern Information Retrieval*," McGray-Hill Int. Book Com., Hamburg, 1983.

[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] 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 Intl. Conf. on Management of Data*, California, USA, 2001.

[13] http://www.connexor.com/demos/index.html.