ON THE TREE INCLUSION AND QUERY EVALUATION IN DOCUMENT DATABASES

Yangjun Chen and Yibin Chen
Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada R3B 2E9

Keywords: Document databases, path-oriented queries, tree inclusion, signatures.

Abstract: In this paper, a method to evaluate queries in document databases is proposed. The main idea of this method is a new top-down algorithm for tree-inclusion. In fact, a path-oriented query can be considered as a pattern tree while an XML document can be considered as a target tree. To evaluate a query $S$ against a document $T$, we will check whether $S$ is included in $T$. For a query $S$, our algorithm needs $O(|T| \cdot \text{height}(S))$ time and no extra space to check the containment of $S$ in document $T$, where $|T|$ stands for the number of nodes in $T$ and height($S$) for the height of $S$. Especially, the signature technique can be integrated into a top-down tree inclusion to cut off useless subtree checkings as early as possible.

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. A lot of work has been done on this issue (GMD, 1992) (C. Zhang, et al., 2001) (INRIA). However, all the methods proposed fail to recognize that the evaluation of a path-oriented query is in essence a tree-inclusion problem. For instance, in (C. Zhang, et al., 2001), a method was proposed to handle the so-called containment queries, which can be considered as a special case of the generic path-oriented queries. The main idea behind it is the inverted indexes, by means of which each element (or a text word) is associated with a set of triples: $(docno, label, level)$, where $docno$ is the document identifier, $label$ is used to indicate the position of an element and to check the containment relationship between elements or between an element and a text word, and $level$ is the level of an element (or a text word) in a document tree. This method works well for single word checkings. However, in the case that a query is a non-trivial tree, its theoretic time complexity is $O(|T|^3)$ where $|T|$ and $|S|$ represent the numbers of nodes in the document tree $T$ and in the query tree $S$, respectively.

In fact, much research has been conducted on the tree-inclusion problem in the theory research community, such as those reported in (W. Chen, 1998) (INRIA) (H. Mannila et al., 1990) (Thorsten Richter, 1997). All the methods focus, however, on the bottom-up strategies to get optimal computational complexities, not suitable for database environment since the algorithms proposed assume that both the target tree (or say, the document tree) and the pattern tree (or say, the query tree) can be accommodated completely in main memory. It is not the case of database applications. In this paper, we propose a top-down algorithm that is of the time complexity comparable to the best bottom-up algorithm (W. Chen, 1998), but needs no extra space overhead. It 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 tree is manipulated. Especially, it can be combined with some kinds of heuristics such as signatures (C. Faloutsos, 1992) to speed-up query evaluation.

The rest of the paper is organized as follows. In Section 2, we discuss the storage structure of XML documents in a relational database. In Section 3, we show that a path-oriented query can be represented as a tree-inclusion problem and discuss our top-down strategy in great detail. Section 4 is devoted to the combination of the signature technique with the tree-inclusion. Finally, a short conclusion is set forth in Section 5.
2 STORAGE OF DOCUMENTS IN DBs

An XML document is defined as having elements and attributes. Elements are always marked up with tags; and an element may be associated with several attributes to identify domain-specific information. XML processors (or parsers) guarantee that XML documents stored in databases follow tagging rules prescribed in XML or conform to a DTD (Document Type Descriptor). Generally, an XML document can be represented as a tree, and node types in the tree are of three kinds: Element, Attribute and Text. These node types are equivalent to the node types in XSL (World Wide Web Consortium, Extensible Style Language (XML) Working Draft, 1998) data model. There are some other less important node types such as comments, processing instructions, etc. The treatment of those node types is trivial and thus will not be discussed here.

- Node type of Element has an element name as the label. Each Element node has zero or more child nodes. The type of each child node is of one of the three types (Element, Attribute and Text).

- Node type of Attribute have an attribute name and an attribute value as a label. Attribute nodes have no child nodes. If there are multiple appearances of attributes, the order of the attributes will be ignored since the attribute order is normally not important for the document treatment.

- Node type of Text have strings as labels. Text nodes have no child nodes, either.

In Fig. 1(b), we show the tree structure representing the XML document shown in Fig. 1(a). To store documents in databases efficiently, the policies shown below should be followed:

- (DTD independent) Database schemas to store XML documents should not depend on DTDs or element types. Any XML document can be manipulated, based on the predefined relations.

- (no loss of structural information) The structure of a document stored in the database should be implemented in some way and can be manipulated.

- (easy maintenance) The cost of the maintenance of the document structure should be kept minimum. Any update to a document will not cause the storage changes of other documents.

To reach above goals, we decompose a document into a set of elements and distribute them over three relations named: Element, Text and Attribute, respectively.

The relation Element has the following structure:

\[ \{ \text{DocID} : \langle \text{integer} \rangle, \text{ID} : \langle \text{integer} \rangle, \text{Ename} : \langle \text{string} \rangle, \text{FirstChildID} : \langle \text{integer} \rangle, \text{SiblingID} : \langle \text{integer} \rangle, \text{AttributeID} : \langle \text{integer} \rangle \} \]

where DocID represents the document identifier, ID represents the element identifier, Ename is the element name (or tag name), firstChildID is the pointer to the first child of an element, siblingID is the pointer to the right sibling of an element, and attributeID is the pointer to the first attribute of an element, which is stored in the relation Attribute.

For example, the document given in Fig. 1(a) can be stored in this table as shown below.

Element:

<table>
<thead>
<tr>
<th>DocID</th>
<th>ID</th>
<th>Ename</th>
<th>FirstChildID</th>
<th>SiblingID</th>
<th>AttributeID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>hotel-room-reservation</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>name%</td>
<td>1</td>
<td>3</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>location</td>
<td>4</td>
<td>11</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>city-or-district%</td>
<td>2</td>
<td>5</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>state%</td>
<td>3</td>
<td>6</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>country%</td>
<td>4</td>
<td>7</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>address</td>
<td>8</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>number%</td>
<td>5</td>
<td>9</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>street%</td>
<td>6</td>
<td>10</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>post-code%</td>
<td>7</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>type%</td>
<td>12</td>
<td>14</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>rooms%</td>
<td>8</td>
<td>13</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>price%</td>
<td>9</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>reservation-time</td>
<td>15</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>from%</td>
<td>10</td>
<td>15</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>to%</td>
<td>11</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Figure 1: A simple document and its tree structure
In the relation Element, an element name suffixed with ‘%’ indicates that its first child is a text appearing in the relation Text. In addition, in the table, ‘*’ represents a null value.

The relation Text has a simpler structure: 
\{DocID: <integer>, textID: <integer>, value: <string>\}, where “textID” is for the identifiers of texts, which are used as the values of the corresponding elements in the original document. One should notice that a text takes always an element as the parent node. See the following table for illustration.

<table>
<thead>
<tr>
<th>TextID</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Travel-lodge</td>
<td>Winnipeg</td>
<td>Manitoba</td>
<td>Canada</td>
<td>500</td>
<td>Portage Ave.</td>
<td>R3B 2E9</td>
<td>one-bed-room</td>
<td>$19 00</td>
<td>April 20, 2002</td>
<td>April 28, 2002</td>
</tr>
</tbody>
</table>

Finally, the relation Attribute has five data fields: 
\{DocID: <integer>, att-ID: <integer>, parentID: <integer>, att-name: <string>, att-value: <string>\}; In the relation Text, we have parentID attribute used for the identifiers of elements (stored in relation “Element”), in which the corresponding attribute appears. The following table helps for a better understanding.

<table>
<thead>
<tr>
<th>docID</th>
<th>att-ID</th>
<th>parentID</th>
<th>Att-name</th>
<th>Att-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>filecode</td>
<td>1302</td>
</tr>
</tbody>
</table>

The method discussed above is quite different from that discussed in (J. Shanmugasundaram et al., 1999), by means of which for each different DTD a different relational schema will be generated. It will obviously increase the heterogeneity of distributed document databases. Considering the web environment, an uniform structure for all the document databases distributed over the network will definitely benefit communication and evaluation of distributed queries.

3 QUERY EVALUATION IN DBs

In this section, we discuss the query evaluation in a document database. First, we show what is a path-oriented query in 3.1. Then, we indicate that the evaluation of path-oriented queries is in essence a tree-inclusion problem, and propose a new top-down algorithm for this task in 3.2.

3.1 Path-oriented queries

Several path-oriented language such as XQL (J. Robie, et al., 1998) and XML-QL (A. Deutsch, et al., 1989) have been proposed to manipulate tree-like XML documents. XQL is a natural extension to the XSL pattern syntax, providing a concise, understandable notation for pointing to specific elements and for searching nodes with particular characteristics. On the other hand, XML-QL has operations specific to data manipulation such as joins and supports transformations of XML data. XML-QL offers tree-browsing and tree-transformation operators to extract parts of documents to build new documents. XQL separates transformation operation from the query language. To make a transformation, an XQL query is performed first, then the results of the XQL query are fed into XSL (World Wide Web Consortium, Extensible Style Language (XML) Working Draft, 1998) to conduct transformation.

An XQL query is represented by a line command which connects element types using path operators (’/’ or ’//’), ’/’ is the child operator which selects from immediate child nodes. ’//’ is the descendant operator which selects from arbitrary descendant nodes. In addition, symbol ’@’ precedes attribute names. By using these notations, all paths of tree representation can be expressed by element types, attributes, ’/’ and ’@’. Exactly, a simple path can be described by the following Backus-Naur Form:

\(<\text{simple path}> ::= <\text{PathOp}> <\text{SimplePathUnit}> | <\text{SimplePathUnit}> @ <\text{AttName}> | <\text{PathOp}> <\text{SimplePathUnit}> @ <\text{AttName}> | <\text{SimplePathUnit}> | <\text{ElementType}> | <\text{PathOp}> <\text{SimplePathUnit}>\).

The following is a simple path-oriented query:

\(\text{/letter/body [para ScontainsSvisited]}\),

where \(\text{/letter//body is a path and [para ScontainsSvisited]}\) is a predicate, enquiring whether element “para” contains a word “visited”. Several paths can be jointed together using ‘\&’ to form a complex query as follows.

\(\text{/hotel-room-reservation/name ?x \& /hotel-room-reservation/location [city-or-district = 'Winnipeg'] \& /hotel-room-reservation/location/address [street = '510 Portage Ave.']}\).

This query will find the name of the hotel located in 510 Portage Ave., Winnipeg.
3.2 Evaluation of Path-Oriented Queries as a Tree Inclusion Problem

Both the documents and the queries can be considered as labeled trees and the evaluation of a path-oriented query can be thought of as a tree-embedding 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 tree representing a query is embedded in a document tree.

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 say 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 shown above can also be represented as a labeled tree.

Definition 2 (tree embedding) Let \( T_1 \) and \( T_2 \) be two labeled trees. A mapping \( M \) from the nodes of \( T_2 \) to the nodes of \( T_1 \) is an embedding of \( T_2 \) into \( T_1 \) if it preserves labels and ancestor-descendant relationship. That is, for all nodes \( u \) and \( v \) of \( T_2 \), we require that

a) \( M(u) = M(v) \) if and only if \( u = v \),

b) label(\( u \)) = label(M(\( u \))), and
c) \( u \) is an ancestor of \( v \) in \( T_2 \) if and only if \( M(u) \) is an ancestor of \( M(v) \) in \( T_1 \).

An embedding is root preserving if \( M(\text{root}(P)) = \text{root}(T) \). According to (Pekka Kilpelainen, et al. 1995), restricting to root-preserving embedding does not lose generality.

Example 1. As an example, consider the trees: \( T \) and \( S \) shown in Fig. 2(a), representing the query shown discussed in 3.1 and the document shown in Fig. 1(a), respectively. If a mapping as shown in Fig. 2(b) can be determined, we’ll have a tree-embedding of the query tree into the document tree. In this case, we say that the query tree is included in the document tree. For the query evaluation purpose, we’ll return that document as one of the answers.

In the following, we discuss a top-down algorithm for tree inclusion, whose computational complexities are comparable to any bottom-up methods for this problem. Especially, we can integrate the signature technique (C. Faloutsos, 1992) into a tree embedding to cut off useless subtree checking, which improves the efficiency significantly. Our algorithm is based on the following three observations:

1. Let \( r_1 \) and \( r_2 \) be the roots of \( T \) and \( S \), respectively. If \( T \) includes \( S \) and label(\( r_1 \)) = label(\( r_2 \)), we must have a root preserving embedding.

2. Let \( T_1, \ldots, T_k \) be the subtrees of \( r_1 \). Let \( S_1, \ldots, S_l \) be the subtrees of \( r_2 \). If \( T \) includes \( S \) and label(\( r_1 \)) = label(\( r_2 \)), there must exist two sequences of integers: \( k_1, \ldots, k_j \) and \( l_1, \ldots, l_j \) (\( j \not= \emptyset \)) such that \( T_{k_i} \) includes \( S_{l_i} \), \( i = 1, \ldots, j \), where \( S_{l_i} \) represents a forest containing subtrees \( S_{l_1}, \ldots, S_{l_j} \) and \( S \) (See Fig. 3 for illustration.)

3. If \( T \) includes \( S \), but label(\( r_1 \)) \not= label(\( r_2 \)), there must exist an \( i \) such that \( T_i \) contains the whole \( S \).
We notice that observation (1) and (3) hint a top-down process to find any possible root-preserving subtree embeddings. However, to work according to observation (2), we will first check \( T_i \) against \( <S_1, \ldots, S_r> \) to find an \( i (1 \leq i \leq l) \) such that \( T_i \) includes \( <S_1, \ldots, S_r> \). If \( i = 0 \), it shows that \( T_i \) does not include any subtree in \( <S_1, \ldots, S_r> \). Next, we will check \( T_{i+1} \) against \( <S_{i+1}, \ldots, S_r> \), and so on. This process can be done in a bottom-up way as discussed below.

Let \( T_{i1}, \ldots, T_{ij} \) be the subtrees of \( T_i \)’s root. To find an \( i \) such that \( T_i \) includes \( <S_1, \ldots, S_r> \), the only way is to check \( T_{ik} \) in turn against \( <S_{i1}, \ldots, S_r> (k = 1, \ldots, j, i_0 = 0) \). It is the same process as indicated by observation (2). That is, if there exists an \( i \) such that \( T_i \) includes \( <S_1, \ldots, S_r> \), then there must exist two sequences of integers: \( e_1, \ldots, e_s \) and \( l_1, \ldots, l_s (s \leq i) \) such that \( T_{il} \) includes \( <S_{i1}, \ldots, S_{il}> \) \((h = 1, \ldots, s, i_0 = 0) \). Therefore, all the same applies to the subtrees of the root of any \( T_{i1} \). Therefore, it is a recursive process and in this process, the node checking is actually done from bottom to top. However, this process is interleaved with a top-down process. That is, whenever a subtree in \( T \) is to be checked against a single \( S \), the top-down process will be invoked to find a possible root-preserving subtree inclusion as illustrated in Fig. 4.

\[
\begin{align*}
T_i & \xrightarrow{\text{checking against}} <S_1, \ldots, S_r> \\
T_{i+1} & \xrightarrow{\text{checking against}} <S_1, \ldots, S_r>(i \leq l) \\
T_{i1} & \xrightarrow{\text{checking against}} <S_r>
\end{align*}
\]

Figure 4: Illustration for calling top-down process

In Fig. 4, we show how \( T_i \) is checked against \( <S_1, \ldots, S_r> \). In the figure, \( T_{i+1}^{(s)} \) stands for a sequence containing \( k \) 1s, and then \( T_{i+1}^{(s)} \) represents the leftmost subtree of \( T_{i+1} \)’s root. For instance, \( T_{i+1}^{(2)} \) (i.e., \( T_{111} \)) is the leftmost subtree of the root of \( T_{11} \) (i.e., \( T_{11} \)). When we check \( T_i \) against \( <S_1, \ldots, S_r> \), \( T_{i+1} \) (i.e., \( T_{i+1}^{(s)} \)) against \( <S_r> \) but \( |T_{i+1}^{(s)}| < |S_r| \), then we will check \( T_{i+1}^{(s)} \) against \( <S_1, \ldots, S_r> \). When we do this, the same method applies. We repeat this process until we meet \( T_{i+1}^{(s)} \) for some \( k \) such that \( |S_k| \leq |T_{i+1}^{(s)}| < |S_k| \). In this case, we will check \( T_{i+1}^{(s)} \) against \( S_k \) in a top-down fashion as discussed above. If \( T_{i+1}^{(s)} \) includes \( S_k \), we will try to check whether \( T_{i+1}^{(s)} \), the direct right sibling subtree of \( T_{i+1}^{(s)} \), includes \( S_{k}, \ldots, S_r \). For some \( j \) such that \( |S_{k}, \ldots, S_r| \leq |T_{i+1}^{(s)}| < |S_{k}, \ldots, S_r| \). Otherwise, we will check whether \( T_{i+1}^{(s)} \), the direct right sibling subtree of \( T_{i+1}^{(s)} \), includes \( S_{k}, \ldots, S_r \) for some \( h \) such that \( |S_{k}, \ldots, S_r| \leq |T_{i+1}^{(s)}| < |S_{k}, \ldots, S_r| \). Obviously, the whole computation is a top-down process with the bottom-up checkings interleaved. Concretely, the top-down and the bottom-up processes are mixed as follows.

- Let \( T' \) be a subtree of \( T \). If there exists an \( i (1 \leq i) \) such that \( |S_1, \ldots, S_r| \leq |T'| < |S_1, \ldots, S_r| \), we will check \( T' \) against \( <S_1, \ldots, S_r> \) in a bottom-up way. That is, we will first check whether the subtrees of \( T' \) include \( <S_1, \ldots, S_r> \).
- If \( |S_1, \ldots, S_r| \leq |T'| < |S_1, \ldots, S_r| \), we will check \( T' \) against \( S_1 \) top-down, by which we will first compare the root of \( T' \) and the root of \( S_1 \).

Since the top-down and bottom-up processes are mixed, we need to find a way to distinguish them. Consider the recursive call to check \( T_{i+1}^{(s)} \) against \( <S_1, \ldots, S_r> \) illustrated in Fig. 4. If the return value is 0, it shows that the subtrees of \( T_{i+1}^{(s)} \)’s root does not contain any subtree in \( <S_1, \ldots, S_r> \). However, \( T_{i+1}^{(s)} \) itself may includes \( S_1 \). So we need to check \( T_{i+1}^{(s)} \) against \( S_1 \) once again. Now, we consider the recursive call to check \( T_{i+1}^{(s)} \) against \( <S_1, \ldots, S_r> \) illustrated in Fig. 4. In this case, both \( T_{i+1}^{(s)} \) and \( <S_1, \ldots, S_r> \) are trees. If the return value is 0, it shows that \( T_{i+1}^{(s)} \) itself does not include \( S_1 \). Then, a second checking as above is not needed. To avoid such a second checking, we mark the root of \( T_{i+1}^{(s)} \) when it is checked against the root of \( S_1 \).

In terms of the above observation, we devise a computation process as below. First of all, in the case of label \( v \) such that \( \text{label}(v) >\text{label}(u) \), we will check whether \( T_i \) includes \( <S_1, \ldots, S_r> \). The process returns an integer \( i \), indicating that \( T_i \) includes \( <S_1, \ldots, S_r> \). If \( i = 0 \), then we will check whether \( T_i \) includes \( <S_1, \ldots, S_r> \) in a next step. If \( i = 0 \), it shows that no subtrees of \( T_i \)’s root includes any subtrees in \( <S_1, \ldots, S_r> \). In this case, we need to check whether \( T_i \) includes \( S_1 \). It is because although no subtrees of \( T_i \)’s root includes any subtrees in \( <S_1, \ldots, S_r> \), \( T_i \) itself may include \( S_1 \). If \( T_i \) includes \( S_1 \), \( i \) will be changed to 1; otherwise, it remains 0. However, if the root of \( T_i \) does not match the root of \( S_1 \), we know that \( T_i \) cannot include \( S_1 \) since in this case we will have to check the subtrees of \( T_i \)’s root against \( S_1 \); and we have already done that with the result \( i = 0 \). We repeat this process until we find a \( k \) such that \( T_k \) contains all the remaining subtrees of \( r_2 \), or find that such a \( k \) does not exist.

In the following algorithm \( \text{tree-inclusion}(T, S) \) is a tree and \( S \) is a tree or a forest. If \( S \) is a forest, a virtual root for it is constructed, which matches any label. Thus, we will actually check the subtrees of \( T \)’s root against the subtrees in \( S \), respectively. In this way, a top-down process is switched over to a bottom-up process. In addition, each node \( v \) in \( T \) is associated with a mark, denoted \( \text{mark}(v) \). If \( v \)’s label is checked against the label of a node \( S_i \), its mark is temporarily set to 1 to avoid possible redundant checkings. But the mark may be dynamically changed in the subsequent execution.
Function \text{tree-inclusion}(T, S) 

Input: $T$ - target tree; $S$ - pattern tree.
Output: 1 if $T$ includes $S$; 0 if $T$ doesn’t include $S$.

begin
  1. if $|T| < |S|$ then \textbf{if} $S$ is a forest: $<S_i, ..., S_j>$
     \textbf{then} $S := <S_i, ..., S_j>$ for some $i$ such that $|S_i, ..., S_j| \leq |T| < |S_i, ..., S_{j+1}|$
  2. \textbf{else} return 0;
  3. let $r_1$ and $r_2$ be the roots of $T$ and $S$, respectively;
  4. \textbf{if} $r_1$ and $r_2$ are not a virtual root \textbf{then} return 0;
  5. \textbf{else} \textbf{if} $S$ is a forest, construct a virtual root $r_2$ for it, which matches any label\(^*\)
  6. \textbf{let} $T_1, ..., T_k$ be the subtrees of $r_1$;
  7. \textbf{let} $S_1, ..., S_l$ be the subtrees of $r_2$;
  8. \textbf{if} label($r_1$) = label($r_2$) \textbf{then} return 1 \underline{else} return 0;
  9. \textbf{while} ($i \leq k$ \textbf{and} \text{temp} \neq \phi) \textbf{do}
     \begin{align*} 
     &10. \quad \{x := \text{tree-inclusion}(T_i, \text{temp});
     &11. \quad \text{if} \ x > 0 \text{ then} \quad \text{temp} := \text{temp} < S_{j+1}, ..., S_j+x>; \\
     &12. \quad \text{else} \quad \{v := \text{tree-inclusion}(T_i, S_j+1); \\
     &13. \quad \text{if} \ v \text{ and } u \text{ have the same label and } \text{mark}(v) = 0 \text{ then return } 1\}; \\
     &14. \quad \text{else} \quad \{\text{mark}(v) := 0\}; \\
     &15. \quad \{\text{mark}(v) \text{ is used only once in this case. Afterwards, it will be set to 0 for the subsequent computation}\}; \\
     &16. \quad \{i := i + 1; j := j + x;\} \\
     &17. \quad \text{if} \ \text{temp} \neq \phi \text{ then} \quad \{\text{if } r_2 \text{ is a virtual root then return } j\}; \\
     &18. \quad \text{else return } 0;\}
     \end{align*}
  19. \textbf{else} \{if $r_2$ is a virtual root then return $l$ \underline{else} return 1;\}
  20. \textbf{else} \{for $i = 1$ to $k$ \textbf{do}
     \begin{align*} 
     &21. \quad \{x := \text{tree-inclusion}(T_i, S_j); \\
     &22. \quad \text{if } x = \text{number-of-trees}(S) \text{ then return 1;\};} \\
     &\text{(* number-of-trees(S) is the number of the trees in S. A tree can be considered as a forest containing only that tree\.*)} \\
     &23. \quad \text{return 0;}\} \\
     \end{align*}
  24. \}
  25. \}
  26. \}
  27. \}
  28. \}
  \}
\end{verbatim}

In Algorithm \text{tree-inclusion}(T, S), line 1 checks whether $|T| < |S|$. If it is the case, the algorithm returns 0 if $S$ is a tree. If $S$ is a forest, we will check $T$ against the first $i$ subtrees such that $<S_i, ..., S_j> \leq |T| < |S_i, ..., S_{j+1}>$ (see line 2). In addition, when we check $T$ against a forest $<S_i, ..., S_j>$, a virtual root for it is constructed, which matches any label. Thus, we will actually check the subtrees of $T$'s root: $T_1, ..., T_k$ against $S_1, ..., S_l$ and $S_1$ to see whether they include $<S_i, ..., S_j>$ (see line 5). This is performed in a while-loop over $T_i$'s. In each step, a recursive call: \text{tree-inclusion}$\left(T_i, <S_{j_1}, ..., S_{j_2}>\right)$ ($i = 1, ..., j$ for some $j$) is carried out, which returns an integer $x$, indicating that $T_i$ includes $<S_{j_1}, ..., S_{j_2}>$ (see line 14). If $x = 0$, i.e., the subtrees of $T_i$'s root do not include any subtree in $<S_{j_1}, ..., S_{j_2}>$, we need to check whether $T_i$ include $S_{j_1}$ since when we check $T_i$ against $<S_{j_1}, ..., S_{j_2}>$ what we have really done is to check the subtrees of $T_i$'s root, not $T_i$ itself (see lines 16 - 19). If $S$ is a tree, the algorithm returns 1 if it is included; otherwise, 0 (see line 22 and 24).

Finally, we note that if the root of $T$ does not match the root of $S$, the algorithm tries to find the first $T_i$ that contains the whole $S$ (see lines 25 - 28).

In addition, we should pay attention to how \text{mark}(v) is used (see lines 10, 17, and 19). Each time when $v$ is checked against a node (not a virtual node) in $S$, \text{mark}(v) is set to 1. It is used to avoid the call \text{tree-inclusion}(T[v], S_1) after \text{tree-inclusion}(T[v], S_{j_1}, ..., S_{j_2}) returns back if $|S_{j_1}| > |T[v]| < |S_{j_1+1}, S_{j_2+2}>$ (see line 17), where $T[v]$ represents a subtree (in $T$ rooted at $v$. It is because in this case \text{tree-inclusion}(T[v], S_1) must have been invoked during the execution of \text{tree-inclusion}(T[v], S_{j_1}, ..., S_{j_2}) and $v$ has been definitely checked against $S_{j_1+1}$'s root in this process, which is recorded by setting \text{mark}(v) to 1 and used to avoid a second checking. However, it is used only in this case. After that, it should be set to 0 again for the rest part of the computation. This arrangement is correct because during the execution of \text{tree-inclusion}(T[v], S_{j_1}, ..., S_{j_2}), if $|S_{j_1}| \leq |T[v]| < |S_{j_1+1}, S_{j_2+2}>$, $v$ itself will be checked against $S_{j_1+1}$'s root. If $|T[v]| > |S_{j_1+1}, S_{j_2+2}>$ for some $i > 1$, we will check the subtrees of $v$ against $<S_{j_1+1}, ..., S_{j_2}>$ and $v$ is not really checked. In addition, in the rest part of the execution of \text{tree-inclusion}(T[v], S_{j_1+1}, ..., S_{j_2}), $v$ is not checked. So, upon the return of \text{tree-inclusion}(T[v], S_{j_1+1}, ..., S_{j_2})>$, we check the value of \text{mark}(v) to see whether \text{tree-inclusion}(T[v], S_{j_1+1}) has been invoked. Obviously, after this checking, \text{mark}(v) should be set to 0 again for the subsequent computation.

Finally, we can show that the time complexity of the algorithm is bounded by $O(|T| \cdot \text{height}(S))$. It is because although a node in $T$ may be checked more than once, it is checked against different nodes in $S$, and all those nodes in $S$ are on a same path. It is also easy to see that the algorithm needs no extra space.

In the following, we apply the algorithm to the trees shown in Fig. 5 and trace the computation step-by-step for a better understanding.

Example 2. Consider two ordered, labeled trees $T$ and $S$ shown in Fig. 5, where each node in $T$ is identified with $t_i$ such as $l_0, l_1, l_1$, and so on; and
each node in $S$ is identified with $s_i$. In addition, each subtree rooted at $t_i(s_i)$ is represented by $T_i(S_i)$.

In the following step-by-step trace, $i_k$ is used as an index variable for scanning the subtrees of $T_i$’s root; $j_k$ is used to scan the corresponding subtrees in $S$; and $x_k$ is used as a temporary variable.

Figure 5: Two trees

<table>
<thead>
<tr>
<th>Step-by-step trace:</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree-inclusion($T, S$)</td>
<td>Call tree-inclusion($T, S$)</td>
</tr>
<tr>
<td>label($t_0$) = label($s_0$)</td>
<td>Check $i_0$ against $s_0$. $i_0$ is for scanning the subtrees of $t_0$; $j_0$ is used to record how many subtrees of $s_0$ is included; $x_0$ is a temporary variable.</td>
</tr>
<tr>
<td>$i_0 := 1; j_0 := 0; x_0 := 0$</td>
<td>recursive call tree-inclusion($T_0, &lt;S_0, S_0&gt;$).</td>
</tr>
<tr>
<td>tree-inclusion($T_0, &lt;S_0, S_0&gt;$)</td>
<td>Check $t_0$ against a virtual root. It always succeeds.</td>
</tr>
<tr>
<td>label($t_0$) = label(virtual-root)</td>
<td>$i_0$ is for scanning the subtrees of $t_0$; $j_0$ is used to record how many subtrees of $s_0$ is included; $x_0$ is a temporary variable. recursive call tree-inclusion($T_0, &lt;S_0, S_0&gt;$).</td>
</tr>
<tr>
<td>$i_0 := 1; j_0 := 0; x_0 := 0$</td>
<td>compare the sizes of $T_0$ and $&lt;S_0, S_0&gt;$:</td>
</tr>
<tr>
<td>tree-inclusion($T_0, S_0$)</td>
<td>since $&lt;S_0, S_0&gt;$ is larger than $T_0$, remove $S_0$ from $&lt;S_0, S_0&gt;$</td>
</tr>
<tr>
<td>label($t_0$) = label($s_0$)</td>
<td>Check $t_0$ against $s_0$. Mark $t_0$.</td>
</tr>
<tr>
<td>return 1</td>
<td>it returns 1, indicating that $T_0$ includes $S_0$.</td>
</tr>
<tr>
<td>$x_1 := 1; j_1 := 1; i_1 := 2$</td>
<td>$i_1$ is increased by 1; $x_1$ is equal to 1 and then $j_1$ is increased by 1. recursive call tree-inclusion($T_0, S_0$).</td>
</tr>
<tr>
<td>tree-inclusion($T_1, S_0$)</td>
<td>Check $t_1$ against $s_2$. Mark $t_1$.</td>
</tr>
<tr>
<td>label($t_2$) ≠ label($s_2$)</td>
<td>it returns 0, indicating that $T_1$ does not include $S_0$. The mark of $t_1$ will prevent the second checking of $T_1$ against $S_0$.</td>
</tr>
<tr>
<td>return 0</td>
<td>$i_1$ is increased by 1; $x_1$ is equal to 1 and then $j_1$ is not increased.</td>
</tr>
<tr>
<td></td>
<td>it returns 1, indicating that $T_1$ includes $S_0$.</td>
</tr>
<tr>
<td></td>
<td>$i_1$ is increased by 1; $x_1$ is equal to 1 and then $j_1$ is increased by 1. recursive call tree-inclusion($T_1, S_0$).</td>
</tr>
<tr>
<td>$i_2 := 1; j_2 := 0; x_2 := 0$</td>
<td>recursive call tree-inclusion($T_1, S_0$).</td>
</tr>
<tr>
<td>tree-inclusion($T_2, S_0$)</td>
<td>Check $t_2$ against $s_2$. Mark $t_2$.</td>
</tr>
<tr>
<td>label($t_2$) = label($s_2$)</td>
<td>it returns 0, indicating that $T_2$ does not include $S_2$. The mark of $t_2$ will prevent the second checking of $T_2$ against $S_2$.</td>
</tr>
<tr>
<td>return 0</td>
<td>$i_2$ is increased by 1; $x_2$ is equal to 0 and then $j_2$ is not increased. recursive call tree-inclusion($T_2, S_0$).</td>
</tr>
<tr>
<td>$x_2 := 0; j_2 := 0; i_2 := 2$</td>
<td>Check $t_2$ against $s_2$. Mark $t_2$.</td>
</tr>
<tr>
<td>tree-inclusion($T_2, S_2$)</td>
<td>it returns 1, indicating that $T_2$ includes $S_2$.</td>
</tr>
<tr>
<td>label($t_2$) = label($s_2$)</td>
<td>$i_2$ is increased by 1; $x_2$ is equal to 0 and then $j_2$ is not increased. recursive call tree-inclusion($T_2, S_2$).</td>
</tr>
<tr>
<td>return 1</td>
<td>it returns 1, indicating that $T_2$ includes $S_2$.</td>
</tr>
<tr>
<td>$x_2 := 0; j_2 := 0; i_2 := 2$</td>
<td>$i_2$ is increased by 1; $x_2$ is equal to 1 and then $j_2$ is not increased by 1. recursive call tree-inclusion($T_2, S_2$).</td>
</tr>
<tr>
<td>return 2</td>
<td>it returns 2, tree-inclusion($T_2, S_0$) returns 2.</td>
</tr>
<tr>
<td>return 1</td>
<td>it returns 1, indicating that $T$ includes $S$.</td>
</tr>
</tbody>
</table>

4 INTEGRATION OF SIGNATURES INTO TREE INCLUSION

An advantage of the top-down strategy is that we can integrate the signature technique into the tree inclusion to speed up the computation. We assign each node $v$ in $T$ a bit string $s_v$, called a signature, and each node $u$ in $S$ a bit string $s_u$ in such a way that if $s_v$ matches $s_u$ then the subtree $T_v$ rooted at $v$ may includes the subtree $S_u$ rooted at $u$. Otherwise, $T_v$ definitely does not contain $S_u$ and the corresponding tree inclusion checkings can be cut off. Here, by “matching”, we mean for each bit set to 1 in $s_v$, the corresponding bit in $s_u$ is also set to 1 while for a bit set to 0 in $s_v$, the corresponding bit in $s_u$ can be 0 or 1.
To do this, we first assign a signature to each label by using a hash function as shown in (C. Faloutsos, 1992). Then, the signature for each node in a labeled tree can be done as follows:

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, let \( v_1, \ldots, v_n \) be its children, then \( s_v = s \lor S_{v_1} \lor \ldots \lor S_{v_n} \), where \( s \) represents the signature for the label associated with \( v \), and \( S_{v_1}, \ldots, S_{v_n} \) are the signatures of \( v_1, \ldots, v_n \), respectively.

**Example 3.** Consider the tree shown in Fig. 6(a). If the signatures assigned to the labels are those shown in Fig. 6(b). Each node in the tree will have a signature as shown in Fig. 6(c).

Given two ordered, labeled trees \( T \) and \( S \), we assign the signatures to their nodes in the same way. During the checking whether \( T \) includes \( S \), we can use signatures to cut off some subtrees of \( T \), which cannot contain \( S \). For this purpose, we change the algorithm tree-inclusion() by introducing the signature checkings into it. The following algorithm is almost the same as the algorithm tree-inclusion( ); but each time when we check whether a subtree of \( T \) includes a subtree of \( S \), the corresponding signatures will be first checked. Of course, before the execution of the algorithm, the node signatures have to be established for both \( T \) and \( S \).

\begin{align*}
T &:\quad a \quad b \\
&\quad l_1 \quad l_2 \\
&\quad f \quad t_{12} \\
&\quad e \quad c \quad d \quad t_{22} \\
&\quad b_1 \quad b_2 \\
\end{align*}

\begin{align*}
S &:\quad a: \quad 0101 \quad 0000 \\
b: \quad 0011 \quad 1000 \\
c: \quad 0001 \quad 0101 \\
d: \quad 0100 \quad 1000 \\
e: \quad 1010 \quad 0000 \\
f: \quad 1100 \quad 0000 \\
\end{align*}

\begin{align*}
\text{Algorithm signature-tree-inclusion}(T, S) \\
\text{Input: } T, S \\
\text{Output: } I, \text{if } T \text{ includes } S; \text{ otherwise, } 0. \\
\text{begin} \\
1. \text{if } |T| < |S| \text{ then } \text{if } S \text{ is a forest: } <S_1, \ldots, S_n> \\
2. \quad \text{then } S := <S_1, \ldots, S_n> \text{ for some } i \text{ such that } |<S_1, \ldots, S_i>| \leq |T| < |<S_1, \ldots, S_i>|+1> \\
3. \quad \text{else return } 0; \text{ } \\
4. \text{let } r_1 \text{ and } r_2 \text{ be the roots of } T \text{ and } S, \text{ respectively; } \\
5. \quad (\text{if } S \text{ is a forest, construct a virtual root for it, which matches any label } s) \\
6. \text{let } r \text{ and } s \text{ be the signatures of } T \text{ and } S, \text{ respectively; } \\
7. \quad \text{if } s \text{ does not match } r \text{ then returns } 0; \text{ } \\
8. \text{let } T_1, \ldots, T_k \text{ be the subtrees of } r; \text{ } \\
9. \text{let } S_1, \ldots, S_l \text{ be the subtrees of } r_2; \text{ } \\
(* \text{the rest part of the algorithm is exactly the same as lines 8 – 28 in Algorithm tree-inclusion( ).} *) \\
\text{end} \\
\end{align*}

We pay attention to line 5. If \( S \) is a forest, a virtual root for \( S \) will be constructed and it does not have a signature. However, its signature can be easily established by superimposing the signatures of all the subtrees in \( S \). Then, in lines 6 and 7, we check the corresponding signatures to remove the checkings for impossible tree inclusion.

**Example 4.** Consider the tree \( T \) and \( S \) shown in Fig. 5 again. To check whether \( T \) includes \( S \), we will assign signatures to the labels and the nodes in \( T \) and \( S \) in the same way as shown in Fig. 7. Assume that the assignment of the signatures to the labels is shown in Fig. 6(b). Then, the checking of the forest containing \( s_1 \) and \( s_2 \) (in \( S \)) against the tree rooted at \( t_1 \) (in \( T \)) 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).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{Cutting off subtrees using virtual node.}
\end{figure}

\section{5 CONCLUSION}

In this paper, a new strategy for evaluating path-oriented queries is discussed. The main idea of the query evaluation is a new algorithm for checking the inclusion of a query tree \( S \) in a document tree \( T \), by which a top-down process is interleaved with a bottom-up computation. The algorithm has the time complexity comparable to 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. Obviously, this cannot be achieved using any bottom-up strategy.

**REFERENCES**


R. Cole, R. Hariharan, P. Indyk. Tree pattern matching and subset matching in deterministic O(n log^3 m)


