Path–Oriented Queries and Tree Inclusion Problem

Yangjun Chen
University of Winnipeg, Canada

INTRODUCTION

With the rapid advance of the Internet, management of structured documents such as XML documents has become more and more important (Marchiori, 1998; Suciu & Vossen, 2000). As a simplified version of SGML, XML is recommended by W3C (World Wide Web Consortium, 1998a) as a document description metalanguage to exchange and manipulate data and documents on the WWW. It has been used to code various types of data in a wide range of application domains, including a Chemical Markup Language for exchanging data about molecules, the Open Financial Exchange for swapping financial data between banks and customers, as well as a Geographical Markup Language for searching geographical information (Bosak, 1997; Zhang & Gruenwald, 2001). Also, a growing number of legacy systems are adapted to output data in the form of XML documents.

In recent years, efforts have been made to find an effective way to generate XML structures that are able to describe XML semantics in underlying relational databases (Chen & Huck, 2001; Florescu & Kossmann, 1999; Shanmugasundaram et al., 1999, 2000; Yoshikawa, Amagasa, Shimura, & Uemura, 2001). However, due to the substantial difference between the nested element structures of XML and the flat relational data, much redundancy is introduced; i.e., the XML data is either flattened into tuples containing many redundant elements or has many disconnected elements. Therefore, it is significant to explore a way to accommodate XML documents which is different from the relational theory. In addition, a variety of XML query languages have been proposed to provide a clue to manipulate XML documents (Abiteboul, Quass, McHugh, Widom, & Wiener, 1996; Chamberlin et al., 2001; Christophides, Cluet, & Simeon, 2000; Deutsch, Fernandez, Florescu, Levy, & Suciu, 1988; Robie, Chamberlin, & Florescu, 2000; Robie, Lapp, & Schach, 1998). Although the languages differ according to expressiveness, underlying formalism, and data model, they share a common feature: path-oriented queries. Thus, finding efficient methods to do path matching is very important to evaluation of queries against huge volumes of XML documents.

BACKGROUND

As a path-oriented language, XQL queries are 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:

\[ <\text{simple path}> ::= <\text{PathOP}> <\text{SimplePathUnit}> | <\text{PathOp}><\text{SimplePathUnit}>’@’<\text{AttName}> <\text{PathOp}> ::= ‘/’ | ‘//’ <\text{SimplePathUnit}> ::= <\text{ElementType}> <\text{PathOp}> <\text{SimplePathUnit}> <\text{ElementType}> ::= <\text{AttName}> \]

The following is a simple path-oriented query:

\[ /\text{letter//body} \text{[para $contains$ \text{‘visit’}]} \]

where /letter//body is a path and [para $contains$ ‘visit’] is a predicate, enquiring whether element “para” contains a word “visit.”

Several paths can be jointed together using \( \land \) to form a complex query as follows:

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

EVALUATION OF PATH-ORIENTED QUERIES

In this section, we show different ways to evaluate a path-oriented query. First, we discuss the basic methods used in a database environment. Then a new strategy for
tree-inclusion, which can be embedded into a document database to provide an efficient way to evaluate path-oriented queries, is discussed in great detail.

**QUERY EVALUATION BASED ON INVERSION**

**Inversion on Elements and Words**

There is a lot of work that considers using relational database techniques to store and retrieve XML documents, such as Arnold-Moore, Fuller, Lowe, Thom, and Wilkinson (1995); Florescu and Kossman (1999); and Zhang, Naughton, DeWitt, Luo, and Lohman (2001). Among them, the most representative is the method discussed in Zhang et al. 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 documents containing the word (or the element) and its position within each document. To speed up the query evaluation, the position of a word (or an element) is recorded as follows:

- \((\text{Dno}, \text{Wposition}, \text{level})\) for a text word,
- \((\text{Dno}, \text{Eposition}, \text{level})\) for an element,

where \(\text{Dno}\) is its document number, \(\text{Wposition}\) is its position in the document, and \(\text{level}\) is its nesting depth within the document; \(\text{Eposition}\) is a pair: \(<s, e>\), representing the positions of the start and end tags of an element, respectively. For instance, the document shown in Figure 1(a) is indexed as shown in Figure 1(b). The index for elements is called \(E\)-index and the index for words is called \(T\)-index.

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 \(x.s < x'.s < x.e\). Let \((d'', x'', l'')\) be an index entry for another element \(c\). Then, \(a\) contains \(c\) iff \(d = d''\) and \(x.s < x''.s < x.e < x''.e\). Using these properties, some simple path-oriented queries can be evaluated. For example, to process the query: /hotel-room-reservation/location/[city-or-district = 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 (note that primary keys are italicized):

\[
\begin{align*}
\text{E-index} & : (\text{element}, \text{docno}, \text{begin}, \text{end}, \text{level}) \\
\text{T-index} & : (\text{word}, \text{docno}, \text{wordPosition}, \text{level})
\end{align*}
\]

These 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 nontrivial tree, the evaluation based on these index structures is an exponential time process. To see this, consider the query: /hotel-room-reservation/location/address [street = 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 using this method.

![Figure 1. A sample XML file and its inverted lists](image-url)
Inversion on Paths and Words

The above method is improved by Seo, Lee, and Kim (2003) 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(path, pathID)
- PathIndex(pathID, docno, begin, end)
- Word(word, wordID)
- 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 = Portage Ave.], only two joins are needed. The first join is between the Path and WordIndex relations with the join condition:

\[
\text{Path.path} = '/\text{hotel-room-reservation/location/address/street}' \land \text{Path.pathID} = \text{WordIndex.pathID}
\]

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

\[
R.\text{wordID} = \text{Word.wordID} \land \text{Word.Word} = '\text{Portage Ave.}'
\]

In general, the query evaluation based on such an index structure needs \(l\) joins, where \(l\) is the number of the words appearing in a query. However, such a time improvement is at the cost of memory space since in Path relation the element names are repeatedly stored. Concretely, for a document with \(n\) nodes, the size of the Path relation is on the order of \(O(n^2)\). Therefore, the time complexity of this method is \(O(l \cdot d \cdot n^2)\), where \(d\) represents the average length of paths.

Query Evaluation Based on Tree Inclusion

As pointed out by Mannila and Raiha (1990), the evaluation of path-oriented queries is in essence a tree inclusion problem. For instance, to evaluate query (2), we will check whether there exists a document that contains the tree representing the query (see Figure 2 for illustration).

In the following, we first give a formal definition of tree inclusion. Then, a new algorithm for checking tree inclusion will be discussed.

**Definition 1 (tree inclusion):** Let \(P\) and \(T\) be rooted labeled trees. Let \(V(P) \ (V(T))\) be the set of the nodes in \(P \ (T)\). We define an ordered embedding \((f, P, T)\) as an injective function \(f: V(P) \to V(T)\) such that for all nodes \(v, u \in V(P)\),

1. \(\text{label}(v) = \text{label}(f(v))\); (label preservation condition)
2. \(v\) is an ancestor of \(u\) iff \(f(v)\) is an ancestor of \(f(u)\); (ancestor condition)
3. \(v\) is to the left of \(u\) iff \(f(v)\) is to the left of \(f(u)\); (sibling condition)

For example, the tree representing the query (2) is included in the tree representing the document shown in Figure 1(a) (see Figure 2).

A lot of algorithms have been developed to check tree inclusion, such as Alonso and Schott (1993); Chen (1998); Kilpelainen and Mannila (1995); and Richter (1997). All these methods focus on the bottom-up strategies to get optimal computational complexities, but they are 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 the main memory. In the case of a large volume of data, it is not possible. Here we present a new algorithm by integrating a top-down process into a bottom-up computation, which has the same time complexity as the best bottom-up algorithm but needs no extra space. More importantly, it is more suitable for a database environment since by the top-down process each time only part of the tree is manipulated. Furthermore, it can be combined with word signatures to speed up query evaluation (Chen, 2003).
Path-Oriented Queries and Tree Inclusion Problem

Function \text{tree-inclusion}(T, P, a) (*top-down process*)

Input: \( T \) - a tree, \( S \) - a tree, \( a \) - an integer (at the very beginning, \( a = 0 \))
Output: \( \text{num}, \text{transnum} \) - a pair of integers

\begin{algorithm}
begin
1. let \( r_1 \) and \( r_2 \) be the roots of \( T \) and \( P \), respectively; 
2. let \( T_1, \ldots, T_k \) be the subtrees of \( r_1 \); 
3. let \( P_1, \ldots, P_k \) be the subtrees of \( r_2 \); 
4. if \( \text{label}(r_1) = \text{label}(r_2) \) then
   \begin{enumerate}
   \item \( \text{temp} := \text{<} P_1, \ldots, P_k ; i := 1 ; j := 0 ; b := 0 ; \)
   \item while \( (i \leq k) \cap \text{temp} \neq 0 \) do
   \item \( x := \text{forest-inclusion}(T, \text{temp}, b) ; \)
   \item if \( x \cdot \text{num} > 0 \) then \( \text{temp} := \text{temp} \cdot \text{<} P_1, \ldots, P_k ; j := j + x \cdot \text{num} ; b := 0 ; \)
   \item else if \( (x \cdot \text{subnum} = \text{number of the subtrees of } P_1 \text{'s root and } \text{label}(T_1 \text{'s root}) = \text{label}(P_1 \text{'s root})) \)
   \item then \( \text{temp} := \text{temp} \cdot \text{<} P_1, \ldots, P_k ; j := j + 1 ; b := 0 ; \)
   \item else \( b := x \cdot \text{transnum} ; \)
   \item \( i := i + 1 ; \)
   \item if \( \text{temp} = 0 \) then return \((1, 0) ; \)
   \item else \( \text{if } j = 0 \) then \( \text{for } i = 1 \text{ to } k \) do
   \item \( y := \text{forest-inclusion}(T, P, b) ; b := y \cdot \text{transnum} ; \)
   \item if \( y \cdot \text{num} = 1 \) then return \((1, 0) ; \)
   \item  
   \item \( \text{else } \{ x := \text{forest-inclusion}(T, <P_1, \ldots, P_k> ; 0) ; \text{return } (0, \text{max}(j, a + x \cdot \text{num})) ; \} \}
   \end{enumerate}
end

end

1. Let \( r_1 \) and \( r_2 \) be the roots of \( T \) and \( P \), respectively. If \( T \) includes \( P \) and \( \text{label}(r_1) = \text{label}(r_2) \), we must have a root preserving embedding.
2. Let \( T_1, \ldots, T_k \) be the subtrees of \( r_1 \). Let \( P_1, \ldots, P_k \) be the subtrees of \( r_2 \). If \( T \) includes \( P \) and \( \text{label}(r_1) = \text{label}(r_2) \), there must exist \( k_1, \ldots, k_j \) and \( l_1, \ldots, l_j \) (\( j \leq k \)) such that \( T_i \) includes \( <P_{l_1}, \ldots, P_j> \) (\( i = 1, \ldots, j ; l_0 = 0 \)), where \( <P_{l_1}, \ldots, P_j> \) represents a forest containing subtrees \( P_{l_1}, \ldots, P_j \), and \( P_i \).
3. If \( T \) includes \( S \), but \( \text{label}(r_1) \neq \text{label}(r_2) \), 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 while observation (2) hints a bottom-up process to find the left embedding for subforest inclusion. During the top-down process, the bottom-up process will be invoked to find the left embedding of \( <P_1, \ldots, P_j> \) in \( <T_1, \ldots, T_k> \); and during the bottom-up process, the top-down process may be invoked to find any possible root-preserving subtree embedding. (See Kilpelainen & Mannila, 1995, for the definitions of the root-preserving embedding and left embedding.)

In a forest \( <T_1, \ldots, T_k> \), the trees \( T_1, \ldots, T_k \) are called the sibling trees of \( T \).

Let \( P_{i_1}, \ldots, P_{i_q} \) be the subtrees of \( P \)’s root. The top-down process is designed as a function \( \text{tree-inclusion}(T, P, a) \), where \( 0 \leq a \leq l \). If \( a > 0 \), it indicates that the left sibling trees of \( T \) cover \( P_{i_1}, \ldots, P_{i_q} \). If \( T \) does not have any left sibling trees or the left sibling trees of \( T \) don’t cover any of \( P_{i_1}, \ldots, P_{i_q} \), \( a \) is set to \( 0 \). The output of \( \text{tree-inclusion}(T, P, a) \) is a pair of the form \( \text{(num}, \text{transnum}) \), where \( \text{num} \in \{0, 1\} \) and \( a \leq \text{transnum} \leq q \). If \( \text{num} = 1 \), it shows that \( T \) includes \( P \). In this case, the value of \( \text{transnum} \) is not important and will be ignored. If \( \text{num} = 0 \), it shows that \( T \) does not include \( P \) but \( T \) covers \( P_{i_1}, \ldots, P_{i_q} \); and thus \( T \), together with its left sibling trees, covers \( P_{i_1}, \ldots, P_{i_q} \).

The bottom-up process is designed as a function \( \text{forest-inclusion}(T, G, a) \), where \( G = <P_{i_1}, \ldots, P_{i_q}> \) is a forest and \( 0 \leq a \leq q \). If \( a > 0 \), it indicates that the left sibling trees of \( T \) cover \( P_{i_1}, \ldots, P_{i_q} \). If \( T \) does not have any left sibling trees or the left sibling trees of \( T \) don’t cover any of \( P_{i_1}, \ldots, P_{i_q} \), \( a \) is equal to \( 0 \). The output of \( \text{forest-inclusion}(T, G, a) \) is a triplet of the form \( \text{(num}, \text{subnum}, \text{transnum}) \), where \( 0 \leq \text{num} \leq s, 0 \leq \text{subnum} \leq q, \) and \( a \leq \text{transnum} \leq q \). If \( \text{num} > 0 \), \text{subnum} and \text{transnum} are not important and can be ignored. If \( \text{num} = 0 \), \text{subnum} indicates that \( T \) covers \( P_{i_1}, \ldots, P_{i_q} \); and \( P_{i_1}, \ldots, P_{i_q} \) if \( \text{subnum} > 0 \); and \text{transnum} indicates that \( T \), together with all its left sibling trees, covers \( P_{i_1}, \ldots, P_{i_q} \); and
Path-Oriented Queries and Tree Inclusion Problem

Function forest-inclusion(S, G, b)
Input: S - a tree, G - a forest, b - an integer
Output: (num, subnum, transnum) - a triple of integers
begin
1. let r be the root of S; let S₁,..., Sₖ be the subtrees of r;
2. let G = S₁,..., Sₖ;
3. find l such that |<S₁,..., Sₖ>| ≤ |S| ≤ |<S₁,..., Sₖ>| => l; return (l, l, l);
4. if l = 1 and height(T₁) ≥ height(P₁) then { x := tree-inclusion(T, P, b); return (x,num, 0, x,transnum); }
5. if (l = 1 and height(T₁) < height(P₁)) or (l = 0) then { let P₁,..., P₉ be the subtrees of P₁; x := forest-inclusion(S, <P₁₁,..., P₉₉>, b); return (0, 0, b + x.num); }
6. if l > 1 then { temp := <P₁,..., P₉>; i := 1; j := 0; c := 0;
7. while (i ≤ k ∧ temp ≠ φ) do
8. { x := forest-inclusion(S, temp, c);
9. if x.num > 0 then {temp := temp< P₉₊₁,..., P₉₊₉>; j := j + x.num; c := 0;}
10. else if (x.subnum = number of the subtrees of P₁'s root and label(T₁'s root) = label(P₁'s root))
11. then {temp := temp< P₉₊₁,..., P₉₊₉>; j := j + 1; c := 0;}
12. else c := x.transnum;
13. i := i + 1; }
14. if (j > 0) then return (i, 0, 0)
15. else if b = 0 then return (0, c, c);
16. else {x := forest-inclusion(S, <P₁₁,..., P₉₉>, l > 0); return (0, c, max{c, b + x.num})}
end

We notice that the meaning of subnum and transnum are quite different. In the case that num = 0, subnum can be used to check whether T includes P by comparing T’s root and P’s root as well as subnum and the number of P’s children. If T’s root and P’s root have the same label and subnum and the number of P’s children are equal, we have T including P. In this way, a repeated checking of T against P can be avoided. On the other hand, transnum is used to avoid the checking of the tree rooted at T’s parent against P if it exists. Let v be the parent of T. Let S be the subtree rooted at v and S₁,..., Sₖ be the subtrees of v with T = Sᵢ for some i. Assume that Pᵢ is not included by any of S₁,..., Sₖ. Then, the return value of each g(Sᵢ, G, a) (i = 1,..., k) is of the form (0, subnumᵢ, transnumᵢ) and records the number of the subtrees in <P₁,..., P₉>, which are covered by S₁,..., Sₖ. Thus, if label(v) = label(rₙ) and transnumᵢ = q, we know that S includes Pᵢ.

First, we give the formal description of the top-down process.

The above algorithm is made up of two parts. The first part contains lines 1-18. The second part contains lines 19-25. In the first part, we handle the case that label(rₙ) = label(rₙ). In this case, we will perform a series of forest-inclusion(T, G, a) (i = 1,..., g for some g ≤ k), where Gᵢ = < P₁,..., P₉ > with l₁ ≤ l₂ ≤ ... ≤ lₙ ≤ l₉ ≤ s and a₁ = 0 ≤ a₁ ≤ ... ≤ aₙ ≤ q (q is the number of the children of Pᵢ’s root.) The return value of each forest-inclusion(T, G, a₁), is a triplet (num, subnum, transnum), according to which Gᵢ and aᵢ are determined for a next call - forest-inclusion(Tᵢ₊₁, Gᵢ, aᵢ) as follows:

(1) If num > 0, then Gᵢ = < P₁,..., P₉ >, lᵢ₊₁ = lᵢ + num, and aᵢ₊₁ = 0. (See line 8.)
(2) If num = 0, subnum = the number of the children of Pᵢ’s root and label(Tᵢ’s root) = label(Pᵢ’s root), then Gᵢ = < P₁,..., P₉ >, lᵢ₊₁ = lᵢ + 1, and aᵢ₊₁ = 0. (See lines 9-10.)
(3) If num = 0, and subnum ¹ the number the children of Pᵢ’s root or label(Tᵢ’s root) ¹ label’s root), then Gᵢ = Gᵢ, lᵢ₊₁ = lᵢ, and aᵢ₊₁ = transnum. (See line 11.)

When all Tᵢ’s are checked, we will determine the return value of tree-inclusion(T, P, a). We distinguish between two cases:

(1) Let j be the number of subtrees in G, which are covered by <T₁,..., Tₙ>. If j = s, return (num, transnum) = (1, 0), indicating that T includes P. In this case, transnum is not important and is simply set to 0. (See line 13.)
(2) If j < s, the return value is (0, transnum), where transnum is determined as follows:
(a) If \( j = 0 \), we will check whether \( b = \) number of the subtrees of \( P_1 \)'s root and \( \text{label}(T_1) = \text{label}(P_1) \)'s root). If it is the case, set \( j \) to 1; otherwise, \( j \) remains 0. (See line 15.)

(b) If \( j > 0 \), we will check whether \( a = 0 \). If it is the case, return \( (0, j) \). Otherwise, we will call for-
est-inclusion \((T, <P_1, ..., P_q>, 0)\). Assume that the return value is \((\text{num'}, \text{subnum'}, \text{transnum'})\). Then, the value of \( \text{transnum} \) is set to \( \max[j, a + \text{num'}] \).

The second part handles the case that \( \text{label}(r_1) \) \( \text{label}(r_2) \). In this case, we will check each \( T_i \) (\( i = 1, ..., k \)) in turn to find any \( T_i \) which includes the whole \( P_1 \).

From the above analysis, we can see that if each time forest-inclusion \((T, G, a)\) returns a correct value, the result returned by tree-inclusion \((T, P, a)\) must be correct. Now we discuss forest-inclusion \((T, G, a)\) in detail. The following is its main idea:

1. Let \( G = <P_1, ..., P_q> \). Find a \( j \) such that \( |T| \geq |<P_1, ..., P_q>| \) but \( |T| < |<P_1, ..., P_q>| \).
2. If \( j > 1 \), we try to find an embedding of \( <P_1, ..., P_q> \) in the subtrees of \( T_1 \)'s root. Let be \( T_1, ..., T_c \), and \( T_1 \) be the subtrees of \( T_1 \)'s root. Perform a series of forest-inclusion \((T, G, b_k) (k = 1, ..., c \) for some \( c \leq 2)\), where \( G_a = <P_1, ..., P_q> \), with \( l_a = 1 \leq l_a \leq ... \leq l_q \) and \( a_0 \leq a_1 \leq ... \leq a_q \leq q \) (\( q \) is the number of the children of \( P_i \)'s root).
3. If \( j = 1 \), i.e., \( |T| \geq |P_1| \) but \( |T| < |<P_1, P_2>| \), and \( \text{height}(T) \leq \text{height}(P_1) \), call tree-inclusion \((T, P_1, a)\). Assume that its return value is \((\text{num}, \text{transnum})\). Then, the return value of forest-inclusion \((T, G, a)\) is set to be \((\text{num}, 0, \text{transnum})\).
4. If \( j = 0 \) but \( \text{height}(T) < \text{height}(P_1) \), or \( j = 0 \), i.e., \( |T| < |P_1| \), call forest-inclusion \((T, <P_1, ..., P_q>, 0)\). Assume that its return value is \((\text{num}, \text{subnum}, \text{transnum})\). Then, the return value of forest-inclusion \((T, G, a)\) is set to be \((0, 0, \text{transnum})\).

From (2) and (4) shown above, we can see that this process is in essence a bottom-up process. However, it is not a pure bottom-up computation since if the condition in (3) is satisfied, a top-down process will be invoked.

Above is the formal description of the algorithm.

**FUTURE TREND**

Document databases can be considered as well-organised information resources, which can be distributed over the Internet and become accessible to end users through the network. For this purpose, remote query evaluation has to be supported to replace the simple navigation along hyperlinks with the navigation through submitting specific queries. In this way, the search of information will be more efficient and more effective. However, the evaluation of remote queries is obviously more challenging than that of local ones, and much research on this is by all means required. Therefore, this must be one of the most important tasks in the near future.

**CONCLUSION**

In this paper, different methods for evaluating path-oriented queries are discussed. They are the inversion on elements and words (IEW), the inversion on paths and words (IPW), and the method based on a new tree-inclusion algorithm. In general, the IPW method has a better time complexity than the IEW, but it needs more memory space. In contrast, the tree inclusion needs no extra space but shows a better time complexity than both the IPW and the IEW. Especially, the signature technique can be integrated into the new tree-inclusion algorithm to cut off nonrelevant documents or nonrelevant elements as early as possible and improve the efficiency significantly.

**REFERENCES**


**KEY TERMS**

**Containment Query**: Queries that are based on the containment and proximity relationships among elements, attributes, and their contents.

**Document Database**: A database designed for managing and manipulating XML documents or even more generic SGML documents.

**Ordered and Labeled Tree**: Ordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is significant.

**Path-Oriented Query**: Queries that are based on the path expressions including element tags, attributes, and key words.

**Signatures**: A signature is a hash-coded bit string assigned to key words used as indexes to speed up information retrieval.

**Tree Inclusion**: Given two ordered labeled trees $T$ and $S$, the tree inclusion problem is to determine whether it is possible to obtain $S$ from $T$ by deleting nodes. Deleting a node $v$ in tree $T$ means making the children of $v$ become the children of the parent of $v$ and then removing $v$. If $S$ can be obtained from $T$ by deleting nodes, we say that $T$ includes $S$.

**XML Document**: A document consisting of an (optional) XML declaration, followed by either an (optional) DTD or XML schema and then followed by document elements.

**XML Schema**: An alternative to DTDs. It is a schema language that assesses the validity of a well-formed element and attribute information items within an XML document. There are two major schema models: W3C XML Schema and Microsoft Schema.