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 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:

<simple path> ::= <PathOP> <SimplePathUnit> | <PathOp><SimplePathUnit>‘@’<AttName>
<PathOp> ::= ‘/’ | ‘//’
<SimplePathUnit>::=<ElementType> | <ElementType><PathOp><SimplePathUnit>

The following is a simple path-oriented query:

/letter//body [para $contains$‘visit’] (1)

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 ∧ to form a complex query as follows:

/hotel-room-reservation/name ?x ∧ (2)

/hotel-room-reservation/location [city-or-district = Winnipeg] ∧
/hotel-room-reservation/location/address [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:

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

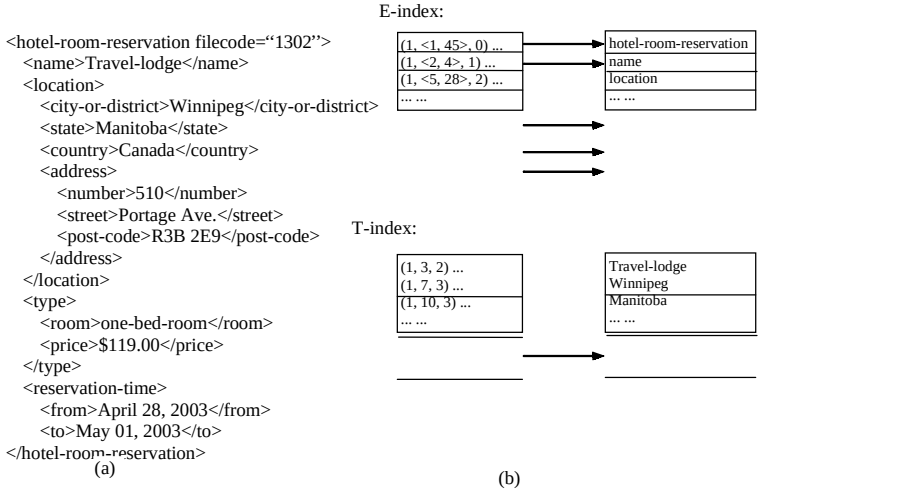
where *Dno* is its document number, *Wposition* is its position in the document, and *level* is its nesting depth within the document; *Eposition* is a pair: <*s*, *e*>, representing the positions of the start and end tags of an element, respectively. For instance, the document shown in Figure 1(a) is 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*' < *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* and *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):

E-index (<u>element</u>, <u>docno</u>, <u>begin</u>, <u>end</u>, <u>level</u>)
T-index (<u>word</u>, <u>docno</u>, <u>wordPosition</u>, <u>level</u>)

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



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:

Path.path = '/hotel-room-reservation/location/address/street' ∧
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 ∧
Word. Word = 'Portage Ave.'

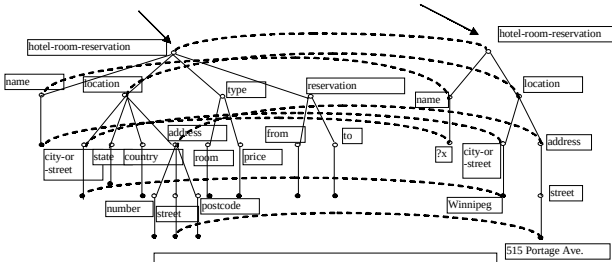
In general, the query evaluation based on such an index structure needs *l* joins, where *l* is the number of the words appearing in a query. However, such a time improvement is at the cost of memory space since in Path relation the element names are repeatedly stored. Concretely, for a document with *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.

Figure 2. Illustration for tree inclusion



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 $f: V(P) \rightarrow V(T)$ such that for all nodes $v, u \in V(P)$,

1. label(*v*) = 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).

Mixed Strategy for Tree Inclusion Problems

Now, we present our algorithm, which is designed based on the following three observations:

```
Function tree-inclusion(T, P, a)                    (*top-down process*)
Input: T - a tree, S - a tree, a - an integer (at the very beginning, a is 0)
Output: (num, transnum) - a pair of integers
begin
1.  let r1 and r2 be the roots of T and P, respectively;
2.  let T1, ..., Tk be the subtrees of r1;
3.  let P1, ..., Ps be the subtrees of r2;
4.  if label(r1) = label(r2) then
5.     {temp := <P1, ..., Ps>; i := 1; j := 0; b := 0;
6.      while (i ≤ k ∧ temp ≠ ϕ) do
7.          { x := forest-inclusion(Ti, temp, b);
8.            if x.num > 0 then {temp := temp/<Pj+1, ..., Pj+x.num>; j := j + x.num; b := 0;}
9.            else if (x.subnum = number of the subtrees of Pj+1's root and label(Ti's root) = label(Pj+1's
root))
10.                then {temp := temp/<Pj+1>; j := j + 1; b := 0;}
11.                else b := x.transnum;
12.           i := i + 1; }
13.    if temp = ϕ then return (1, 0);
14.       else { if j = 0 then
15.                          {if (b = number of the subtrees of P1's root and label(T's root) = label(P1's
root)) then j := 1;}
16.             if a = 0 then return (0, j);
17.             else {x := forest-inclusion(T, <Pa+1, ..., Ps >, 0); return (0, max{j, a + x.num});}}
18.   }
19. else {b := 0;
20.       for i = 1 to k do
21.       { y := forest-inclusion(Ti, P, b); b := y.transnum;
22.         if y.num = 1 return (1, 0 );}
23.       if a = 0 then return (0, b);
24.       else {x := forest-inclusion(T, <Pa+1, ..., Ps>, 0); return (0, max{b, a + x.num})}}
25.       }
end
```

1. Let r_1 and r_2 be the roots of T and P, respectively. If T includes P and $label(r_1) = label(r_2)$, we must have a root preserving embedding.
2. Let T_1, ..., and T_k be the subtrees of r_1. Let P_1, ..., and P_l be the subtrees of r_2. If T includes P and $label(r_1) = label(r_2)$, there must exist k_1, ..., k_j and l_1, ..., l_j ($j \le l$) such that T_{k_i} includes $< P_{l_{i-1}+1}, ..., P_{l_i} >$ (i = 1, ..., j; $l_0 = 0$), where $< P_{l_{i-1}+1}, ..., P_{l_i} >$ represents a forest containing subtrees $P_{l_{i-1}+1}$, ..., and P_{l_i}.
3. If T includes S, but $label(r_1) \neq label(r_2)$, 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 $<P_1, ..., P_l>$ in $<T_1, ..., T_k>$; 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 emdedding.)

In a forest $<T_1, ..., T_k>$, the trees T_1, ..., T_{i-1} are called the sibling trees of T_i.

Let P_{11}, ..., and P_{1q} be the subtrees of P_1's root. The top-down process is designed as a function *tree-inclusion*(T, P, a), where $0 \le a \le l$. If $a > 0$, it indicates that the left sibling trees of T cover P_1, ..., and P_a. If T does not have any left sibling trees or the left sibling trees of T don't cover any of P_1, ..., and P_l, a is set to 0. The output of *tree-inclusion*(T, P, a) is a pair of the form (*num*, *transnum*), where *num* $\in \{0, 1\}$ and $a \le$ *transnum* $\le q$. If *num* = 1, it shows that T includes P. In this case, the value of *transnum* is not important and will be ignored. If *num* = 0, it shows that T does not include P but T covers P_a, ..., and $P_{transnum}$; and thus T, together with its left sibling trees, covers P_1, ..., and $P_{transnum}$.

The bottom-up process is designed as a function *forest-inclusion*(T, G, a), where $G = <P_1, ..., P_s>$ is a forest and $0 \le a \le q$. If $a > 0$, it indicates that the left sibling trees of T cover P_{11}, ..., and P_{1a}. If T does not have any left sibling trees or the left sibling trees of T don't cover any of P_{11}, ..., and P_{1q}, a is equal to 0. The output of *forest-inclusion*(T, G, a) is a triplet of the form (*num*, *subnum*, *transnum*), where $0 \le$ *num* $\le s$, $0 \le$ *subnum* $\le q$, and $a \le$ *transnum* $\le q$. If *num* > 0, *subnum* and *transnum* are not important and can be ignored. If *num* = 0, *subnum* indicates that T covers P_{11}, ..., and $P_{1subnum}$ if *subnum* > 0; and *transnum* indicates that T, together with all its left sibling trees, covers P_{11}, ..., and

```
Function forest-inclusion(S, G, b)                    (*bottom-up process*)
Input: S - a tree, G - a forest, b - an integer
Output: (num, subnum, transnum) - a triple of integers
begin
1.   let r1 be the root of S; let S1, ..., Sk be the subtrees of r1;
2.   let G = <P1, ..., Ps>;
3.   find l such that |<P1, ..., Pl>| ≤ |S| < |<P1, ..., Pl+1>|
4.   if l = 1 and height(Ti) ≥ height(P1) then {x := tree-inclusion(T, P1, b); return (x.num, 0,
     x.transnum);}
5.   if (l = 1 and height(Ti) < height(P1)) or (l = 0) then
6.   { let P11, ..., P1q be the subtrees of P1's root;
7.     x : = forest-inclusion(S, <P1,b+1, ..., P1q >, 0);
8.     return (0, 0, b + x.num);}
9.   if l > 1 then
10.  { temp := <P1, ..., Pl>;        i := 1; j := 0; c := 0;
11.    while (i ≤ k ∧ temp ≠ ϕ) do
12.    { x := forest-inclusion(Si, temp, c);
13.      if x.num > 0 then {temp := temp/<Pj+1, ..., Pj+x.num>; j := j + x.num; c := 0;}
14.      else if (x.subnum = number of the subtrees of Pj+1's root and label(Ti's root) = label(Pj+1's root))
15.           then {temp := temp/<Pj+1>; j := j + 1; c := 0;}
16.           else c := x.trannum;
17.      i := i + 1; }
18.  if (j > 0) then return (j, 0, 0)
19.  else { if b = 0 then return (0, c, c);
20.       else {x := forest-inclusion(S, <Pj+1,b+1, ..., Pj+1,l >, 0); return (0, c, max{c, b + x.num)}}}
end
```

$P_{1transnum}$. 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_1 by comparing T's root and P_1's root as well as *subnum* and the number of P_1's children. If T's root and P_1's root have the same label and *subnum* and the number of P_1's children are equal, we have T including P_1. In this way, a repeated checking of T against P_1 can be avoided. On the other hand, *transnum* is used to avoid the checking of the tree rooted at T's parent against P_1 if it exists. Let v be the parent of T. Let S be the subtree rooted at v and S_1, ..., S_k be the subtrees of v with $T = S_i$ for some i. Assume that P_1 is not included by any of S_1, ..., S_k. Then, the return value of each $g(S_i, G, a_i)$ (i = 1, ..., k) is of the form (0, $subnum_i$, $transnum_i$) and $transnum_k$ records the number of the subtrees in $<P_{11}$, ..., and $P_{1q}>$, which are covered by S_1, ..., S_k. Thus, if $label(v) = label(r_1)$ and $transnum_k = q$, we know that S includes P_1.

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 $label(r_1) = label(r_2)$. In this case, we will perform a series of *forest-inclusion*(T_i, G_i, a_i) (i = 1, ..., g for some $g \le k$), where $G_i = < P_{l_i}$, ..., $P_s>$ with $l_1 = 1 \le l_2 \le ... \le l_g \le s$ and $a_1 = 0 \le a_2 \le ... \le a_g \le q$ (q is the number of the children of P_1's root.) The return value of each *forest-inclusion*(T_i, G_i, a_i), is a triplet (*num*, *subnum*, *transnum*), according to which G_{i+1} and a_{i+1} are determined for a next call - *forest-inclusion*(T_{i+1}, G_{i+1}, a_{i+1}) as follows:

(1) If *num* > 0, then $G_{i+1} = < P_{l_i+num}$, ..., $P_s>$, $l_{i+1} = l_i +$ *num*, and $a_{i+1} = 0$. (See line 8.)
(2) If *num* = 0, *subnum* = the number of the children of P_{l_i}'s root and $label(T_i$'s root) = $label($ P_{l_i}'s root), then $G_{i+1} = < P_{l_i+1}$, ..., $P_s>$, $l_{i+1} = l_i + 1$, and $a_{i+1} = 0$. (See lines 9-10.)
(3) If *num* = 0, and *subnum* ¹ the number the children of P_{l_i}'s root or $label(T_i$'s root) ¹ $label($'s root), then $G_{i+1} = G_i$, $l_{i+1} = l_i$, and $a_{i+1} = transnum$. (See line 11.)

When all T_i'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 $<T_1$, ..., $T_k>$. 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 *label*(T's root) = *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 *forest-inclusion*(T, $<P_{a+1}, ..., P_s>$, 0). Assume that the return value is (*num'*, *subnum'*, *transnum'*). Then, the value of *transnum* is set to *max*{j, a + *num'*}.

The second part handles the case that *label*(r_1) ¹ *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.

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

(1) Let $G_i = <P_1, ..., P_s>$. Find a j such that $|T_i| \geq |<P_1, ..., P_j>|$ but $|T_i| < |<P_1, ..., P_{j+1}>|$.

(2) If $j > 1$, we try to find an embedding of $<P_1, ..., P_j>$ in the subtrees of T_i's root. Let be T_{i1}, ..., and T_{iz} be the subtrees of T_i's root. Perform a series of *forest-inclusion*(T_i, G_{ik}, b_k) (k = 1, ..., c for some $c \leq z$), where $G_{ik} = < P_{l_k}, ..., P_s>$, with $l_1 = 1 \leq l_2 \leq ... \leq l_c \leq j$ and $a_1 = 0 \leq a_2 \leq ... \leq a_g \leq q$ (q is the number of the children of P_1's root).

(3) If j = 1, i.e., $|T_i| \geq |P_1|$ but $|T_i| < |<P_1, P_2>|$, and *height*(T_i) ≥ *height*(P_1), call *tree-inclusion*(T_i, P_1, a). Assume that its return value is (*num*, *transnum*). Then, the return value of *forest-inclusion*(T_i, G_i, a_i) is set to be (*num*, 0, *transnum*).

(4) If j = 1 but *height*(T_i) < *height*(P_1), or j = 0, i.e., $|T_i| < |P_1|$, call *forest-inclusion*(T_i, $<P_{1,a+1}, ..., P_1 q >$, 0). Assume that its return value is (*num*, *subnum*, *transnum*). Then, the return value of *forest-inclusion*(T_i, G_i, a_i) is set to be (0, 0, *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

Abiteboul, S., Quass, D., McHugh, J., Widom, J., & Wiener, J. (1996). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, *1*(1).

Alonso, L., & Schott, R. (1993). On the tree inclusion problem. In *Proceedings of Mathematical Foundations of Computer Scienc*e (pp. 211-221).

Arnold-Moore, T., Fuller, M., Lowe, B., Thom, J., & Wilkinson, R. (1995). The ELF data model and SGQL query language for structured document databases. In *Proceedings of the Australasian Database Conference* (pp. 17-26).

Bosak, J. (1997, March). *XML, Java, and the future of the Web*. Retrieved from *http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.html*

Chamberlin, D., Clark, J., Florescu, D., Robie, J., Simeon, J., & Stefanescu, M. (2001). *Xquery 1.0: An XML query language* (Tech. Rep., W3C Working Draft 07). World Wide Web Consortium.

Chen, M. (1998). More efficient algorithm for ordered tree inclusion. *Journal of Algorithms, 26,* 370-385.

Chen, Y. (2003) Query evaluation and Web recognition in document databases. In *Proceedings of IASTED International Conference on Internet and Multimedia Systems and Application, IMSA 2003.*

Chen, Y., & Huck, G. (2001). On the evaluation of path-oriented queries in document databases. In *Lecture notes in computer science: Vol. 2113* (pp. 953-962).

Christodoulakis, S., & Faloutsos, C. (1984). Design consideration for a message file server. *IEEE Transactions on Software Engineering, 10*(2), 201-210.

Christophides, V., Cluet, S., & Simeon, J. (2000). On wrapping query languages and efficient XML integration. *In Proceedings of the ACM SIGMOD Conference on Management of Data* (pp. 141-152).

Deutsch, A., Fernandez, M. F., Florescu, M. D., Levy, A., & Suciu, D. (1988, August). *XML-QL: A query language for XML*. Retrieved from *http://www.w3.org/TR/NOTE-xml-ql/*

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys, 17*(1), 49-74.

Faloutsos, C. (1992). Signature files. In W. B. Frakes & R. Baeza-Yates (Eds.), *Information retrieval: Data structures & algorithms* (pp. 44-65). NJ: Prentice Hall.

Florescu, D., & Kossman, D. (1999). Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin, 22*(3).

Kilpelainen, P., & Mannila, H. (1995). Ordered and unordered tree inclusion. *SIAM Journal of Computing, 24,* 340-356.

Knuth, D. E. (1973). *The art of computer programming: Sorting and searching*. London: Addison-Wesley.

Mannila, H., & Raiha, K.-J. (1990). On query languages for the p-string data model. In H. Kangassalo, S. Ohsuga, & H. Jaakola (Eds.), *Information modelling and knowledge bases* (pp. 469-482). Amsterdam: IOS Press.

Marchiori, M. (1998). *QL'98—Query Languages 1998*). Retrieved from *http://www.w3.org/TandS/QL/QL98*

Pixley, T. (2000). *Document Object Model (DOM) Level 2 Events Specification Version 1.0* (W3C Recommendation).

Richter, T. (1997). A new algorithm for the ordered tree inclusion problem. In *Lecture Notes of Computer Science: Vol. 1264. Proceedings of the eighth annual Symposium on Combinatorial Pattern Matching, CPM* (pp. 150-166). Springer.

Riordan, J. (1968). *Combinatorial identities*. NY: Wiley.

Robie, J., Chamberlin, D., & Florescu, D. (2000). Quilt: An XML query language for heterogeneous data sources. In *Proceedings of the International Workshop on the Web and Databases*.

Robie, J., Lapp, J., & Schach, D. (1998) XML query language (XQL). In *W3C QL'98—The Query Languages Workshop*.

Seo, C., Lee, S., & Kim, H. (2003). An efficient inverted index technique for XML documents using RDBMS. *Information and Software Technology, 45,* 11-22.

Shanmugasundaram, J., Shekita, R., Carey, M. J., Lindsay, B. G., Pirahesh, H., & Reinwald, B. (2000). Efficiently publishing relational data as XML documents. In *Proceedings of the International Conference on Very Large Data Bases, VLDB'00* (pp. 65-76).

Shanmugasundaram, J., Tufte, K., Zhang. C., He, D. J., DeWitt, J., & Naughton, J. F. (1999). Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of The International Conference on Very Large Data Bases, VLDB'99* (pp. 302-314).

Suciu, D., & Vossen, G. (2000). *Lecture notes in computer science: Proceedings of the third International Workshop on the Web and Databases, WebDB 2000*. Springer-Verlag.

World Wide Web Consortium. (1998a, February). *Extensible Markup Language (XML) 1.0*. Retrieved from *http//www.w3.org/TR/ 1998/REC-xml/19980210*

World Wide Web Consortium. (1998b, December). *Extensible Style Language (XML) Working Draft*. Retrieved from *http/ /www.w3.org/TR/1998/WD-xsl-19981216*

Yoshikawa, M., Amagasa, T., Shimura, T., & Uemura, S. (2001). XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology, 1*(1).

Zhang, C., Naughton, J., DeWitt, D., Luo, Q., & Lohman, G. (2001). On supporting containment queries in relational database management systems. In *Proceedings of ACM SIGMOD 2001.*

Zhang, J., & Gruenwald, L. (2001). A GML-based open architecture for building a geographical information search engine over the Internet. In *Proceedings of WISE 2001* (pp. 385-392).

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.