Chapter XV

On the Computation of Recursion in Relational Databases

Yangjun Chen
University of Winnipeg, Canada

ABSTRACT

A composite object represented as a directed graph is an important data structure which requires efficient support in CAD/CAM, CASE, office systems, software management, Web databases and document databases. It is cumbersome to handle such an object in relational database systems when it involves recursive relationships. In this chapter, we present a new encoding method to support the efficient computation of recursion. In addition, we devise a linear time algorithm to identify a sequence of reachable trees (w.r.t.) a directed acyclic graph (DAG), which covers all the edges of the graph. Together with the new encoding method, this algorithm enables us to compute recursion w.r.t. a DAG in time O(e)*, where* e *represents the number of edges of the DAG. More importantly, this method is especially suitable for a relational environment.*

INTRODUCTION

It is a general opinion that relational database systems are inadequate for manipulating composite objects which arise in novel applications such as Web and document databases (Mendelzon, Mihaila & Milo, 1997; Abiteboul et al., 1997; Chen & Aberer, 1998, 1999), CAD/CAM, CASE, office systems and software management (Banerjee et al., 1988; Teuhola, 1996). Especially when recursive relationships are involved, it is cumbersome to handle them in a relational system. To mitigate this problem to some extent, many methods have been proposed, such as *join index* (Valduriez & Borel, 1986) and *clustering of composition hierarchies* (Haskin & Lorie, 1982), as well as the encoding scheme (Teuhola, 1996).

In this chapter, we present a new encoding method to facilitate the computation of recursive relationships of nodes in a DAG. In comparison with Teuhola's, our method is simple and space-economic. Specifically, the problem of Teuhola's so-called *signature conflicts* is removed.

BACKGROUND

A composite object can be generally represented as a directed graph. For example, in a CAD database, a composite object corresponds to a complex design, which is composed of several subdesigns (Banerjee et al., 1988). Often, subdesigns are shared by more than one higher-level design, and a set of design hierarchies thus forms a directed acyclic graph (DAG). As another example, the citation index of scientific literature, recording reference relationships between authors, constructs a directed cyclic graph. As a third example, we consider the traditional organization of a company, with a variable number of manager-subordinate levels, which can be represented as a tree hierarchy. In a relational system, composite objects must be fragmented across many relations, requiring joins to gather all the parts. A typical approach to improving join efficiency is to equip relations with hidden pointer fields for coupling the tuples to be joined (Carey et al., 1990). Recently, a new method was proposed by Teuhola (1996), in which the information of the ancestor path of each node is packed into a fix-length code, called the *signature*. Then, the operation to find the transitive closure w.r.t. a directed graph can be performed by identifying a series of signature intervals. No joins are needed. Using Teuhola's method, CPU time can be improved up to 93% for trees and 45% for DAGs in comparison with a method which performs a SELECT command against each node, where the relation to store edges is equipped with a clustering index on the parent nodes (Teuhola, 1996).

In this chapter, we follow the method proposed in Teuhola (1996), but using a different encoding approach to pack "ancestor paths." For example, in a tree hierarchy, we associate each node v with a pair of integers (α, β) such that if v', another node associated with (α', β'), is a descendant of v, some arithmetical relationship between α and α' as well as β and β' can be determined. Then, such relationships can be used to find all descendants of a node and the recursive closure w.r.t. a tree can be computed very efficiently. This method can be generalized to a DAG or a directed graph containing cycles by decomposing a graph into a sequence of trees (forests), in which the approach described above can be employed. As we will see later, a new method can be developed based on the techniques mentioned above, by which recursion can be evaluated in O(e)

time, just as an algorithm using *adjacency lists*. The adjacency list is a common data structure to store a graph in computational graph theory (see Mehlhon, 1984). However, our method is especially suitable for the implementation in a relational environment. More importantly, the proposed encoding scheme provides a new way to explore more efficient graph algorithms to compute transitive closures.

TASK DEFINITION

We consider composite objects represented by a directed graph, where nodes stand for objects and edges for parent-child relationships, stored in a binary relation. In many applications, the transitive closure of a graph needs to be computed, which is defined to be all ancestor-descendant pairs. A lot of research has been directed to this issue. Among them, the semi-naive (Bancihon & Ramakrishnan, 1986) and the logarithmic (Valduriez & Boral, 1986) are typical *algorithmic* solutions. Another main approach is the materialization of the closure, either partially or completely (Agrawal & Jagadish, 1990). Recently, the implementation of the transitive closure algorithms in a relational environment has received extensive attention, including performance and the adaptation of the traditional algorithms (Abiteboul et al., 1990; Agrawal, Dar & Jagadish, 1990; Ioannidis, Ramakrishnan & Winger, 1993; Dar & Ramakrishnan, 1994; Teuhola, 1996).

The method proposed in this chapter can be characterized as a partial materialization method. Given a node, we want to compute all its descendants efficiently based on a specialized data structure. The following is a typical structure to accommodate part-subpart relationship (Cattell & Skeen, 1992):

- Part(Part-id, Part-rest),
- Connection(Parent-id, Child-id, Conn-rest),

where Parent-id and Child-id are both foreign keys, referring to Part-id. In order to speed up the recursion evaluation, we'll associate each node with a pair of integers which helps to recognize the ancestor-descendant relationships.

In the rest of the chapter, the following three types of graphs will be discussed.

i) Tree hierarchy, in which the parent-child relationship is of one-to-many type, i.e., each node has at most one parent.
ii) Directed acyclic graph (DAG), which occurs when the relationship is of many-to-many type, with the restriction that a part cannot be sub/superpart of itself (directly or indirectly).
iii) Directed cyclic graph, which contains cycles.

Later we'll use the term *graph* to refer to the *directed graph*, since we do not discuss non-directed ones at all.

LABELLING A TREE STRUCTURE

In the method proposed in Teuhola (1996), each node v is associated with an interval (l, h), where l and h *are* two signatures each consisting of a bit string. These bit strings are constructed in such a way that if the interval associated with a descendant of v is $(l',$

h'), then $l \leq l'$ and $h \geq h'$ hold. Although this method is incomparably superior to a trivial method, it suffers from the follows disadvantages:

1) This method is space-consuming since signatures tend to be very long.
2) The size of signatures has to be pre-determined. Different applications may require different signature lengths. This can be tuned only manually.
3) There may exist the so-called signature conflicts, i.e., two nodes may be assigned the same signature.

In the following, we search for remedies for these three drawbacks. First, we discuss a tree labeling method to demonstrate the main idea of the improvement in this section. The discussion on general cases will occur later on.

Consider a tree *T*. By traversing *T* in *preorder*, each node *v* will obtain a number *pre*(*v*) to record the order in which the nodes of the tree are visited. In the same way, by traversing *T* in *postorder*, each node will get another number *post*(*v*). These two numbers can be used to characterize the ancestor-descendant relationship as follows.

Proposition 1. Let *v* and *v*' be two nodes of a tree *T*. Then, *v*' is a descendant of *v* if *pre*(*v'*) > *pre*(*v*) and *post*(*v'*) < *post*(*v*).
Proof. See Knuth (1973).

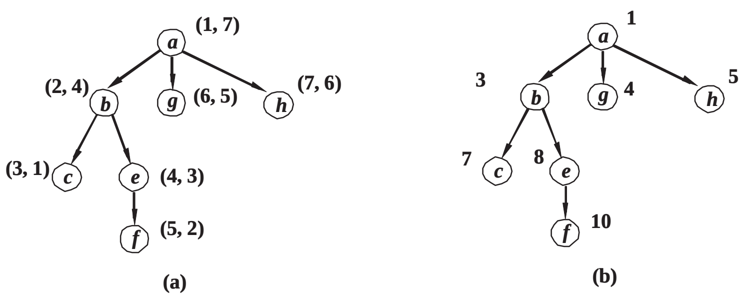
If *v*' is a descendant of *v*, then we know that *pre*(*v'*) > *pre*(*v*) according to the preorder search. Now we assume that *post*(*v'*) > *post*(*v*). Then, according to the postorder search, either *v'* is in some subtree on the right side of *v,* or *v* is in the subtree rooted at *v'*, which contradicts the fact that *v*' is a descendant of *v.* Therefore, *post*(*v'*) must be less than *post*(*v*).

The following example helps for illustration.

Example 1

See the pairs associated with the nodes of the graph shown in Figure 1(a). The first element of each pair is the preorder number of the corresponding node and the second is the postorder number of it. Using such labels, the ancestor-descendant relationship can be easily checked.

Figure 1. Labeling a Tree



For example, by checking the label associated with *b* against the label for *f*, we know that *b* is an ancestor of *f* in terms of Proposition 1. We can also see that since the pairs associated with *g* and *c* do not satisfy the condition given in Proposition 1, *g* must not be an ancestor of *c* and vice versa.

According to this labeling strategy, the relational schema to handle recursion can consist of only one relation of the following form:

Node(Node_id, label_pair, Node_rest),

where label_pair is used to accommodate the preorder number and the postorder number of each node, denoted label_pair.preorder and label_pair.postorder, respectively. Then, to retrieve the descendants of node *x*, we issue two queries. The first query is very simple as shown below:

```
SELECT    label_pair
          FROM   Node
          WHERE Node_id = x
```

Let the label pair obtained by evaluating the above query be *y*. Then, the second query will be of the following form:

```
SELECT    *
          FROM   Node
          WHERE label_pair.preorder > y.preorder
   and    label_pair.postorder < y.postorder
```

From the above discussion, we can see that the three drawbacks of Teuhola's method (Teuhola, 1996) mentioned above can be eliminated: 1) each node is associated with only a pair of integers and therefore the space overhead is low; 2) the size of each label pair remains the same for all applications; 3) there are no signature conflicts since each label pair is different from the others.

In the following, we show two other important techniques to identify the sibling relationship and the parent-child relationship.

For the first task, consider a new labeling method as shown in Figure 1(b). First we assign 1 to the root; then during the breadth-first traversal, we number the children of each node consecutively from $x + 2$, where x is the largest number assigned so far. We call such a labeling method the *sibling-code*. Then, we can associate each parent node with an interval $[a, b]$ such that each child's sibling-code $s \in [a, b]$. Therefore, two nodes are siblings if their sibling-codes belong to the same interval.

To identify the parent-child relation, we associate each node with a level number. The root has the level number 0. All the children of the root have the level number 1, and so on. Then, if node x is the ancestor of y and at the same time $l(x) = l(y) - 1$ ($l(x)$ stands for the level number of x), then we know that x is the parent of y.

GENERALIZATION

Now we discuss how to treat the recursion w.r.t. a general structure: a DAG or a graph containing cycles. First, we address the problem with DAGs. Then, the cyclic graphs will be discussed.

Recursion W.R.T. DAGs

We want to apply the technique discussed above to a DAG. To this end, we define the following concept (Shimon, 1979).

Definition 1. A subgraph $G'(V, E')$ of a finite DAG $G(V, E)$ is called a *branching* if $d_{\text{in}}(v) \leq 1$ for every $v \in V$ ($d_{\text{in}}(v)$ represents the in-degree of v).

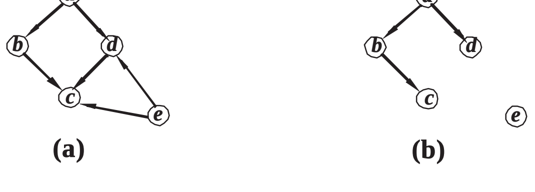
For example, the graph shown in Figure 2(b) is a branching of the graph shown in Figure 2(a).

Let each edge $e \in E$ have a cost $c(e)$. A branching $G'(V, E')$ is a called a *maximal branching* if $\sum_{e \in E'} c(e)$ is maximum. In addition, a tree appearing in a branching and rooted at a node r is called a reachable-tree from r.

In the following, we will divide a DAG into a set of reachable trees. This method shares the flavor of Teuhola's (1996). But our decomposition strategy is quite different from Teuhola's. In his method, a DAG is decomposed into a set of reachable trees which are separated from each other, i.e., there are no common nodes between any two reachable trees, while in ours two reachable trees may have common nodes. The advantage of our method can be seen in the following discussion.

Below we concentrate only on single-root graphs for simplicity. But the proposed method can be easily extended to normal cases. We construct a sequence of reachable trees for a DAG G with the single-root r_0, which covers all the reachable edges from r_0 in G. For our problem, we assume that each edge e in G is associated with a cost $c(e) = 1$. Given r_0, we are interested in the maximal branching B in G, and the reachable tree from r_0 in B, denoted $T_{max}(G)$. First, we recognize $T_{max}(G)$ from G. Then, we remove $T_{max}(G)$ and subsequently all isolated nodes from G, getting another graph G_1. Next, for a leaf node r_1 in $T_{max}(G)$, we construct another reachable tree from r_1 in G_1: $T_{max}(G_1)$ and remove $T_{max}(G_1)$ and all isolated nodes from G_1. Next, for a node r_2, which is a leaf node of $T_{max}(G)$ or $T_{max}(G_1)$, we construct a third reachable tree. We repeat this process until the remaining graph becomes empty. It is therefore easy to see that all $T_{max}(G_i)$'s can be obtained in O(k(n +

Figure 2. A DAG and One of its Brachings



e)) time by repeating graph search procedure *k* times, where *n* and *e* represent the number of the nodes and the edges of the DAG, respectively, and *k* is the number of trees into which *G* can be decomposed. However, this time complexity can be reduced to O($n + e$) by implementing an algorithm which computes such a sequence in a single-scan.

For a DAG $G = (V, E)$, we represent the sequence of reachable trees $T_{max}(G_i)$ ($i = 0, 1, ..., m$; $G_0 = G$) as follows:

$$\begin{aligned}
&T_{max}(G_0) = (V_1, E_1),\\
&T_{max}(G_1) = (V_2, E_2),\\
&T_{max}(G_2) = (V_3, E_3),\\
&\dots\dots\\
&T_{max}(G_m) = (V_{m+1}, E_{m+1}),
\end{aligned}$$

where V_1 stands for the set of nodes in *G*, V_i ($i = 2, ..., m+1$) for the set of nodes in $G - E_1 \cup E_2 \cup ... \cup E_{i-1}$, and *m* is the largest in-degree of the nodes of *G*.

In the following, we give a linear time algorithm to compute all $T_{max}(G_i)$'s.

The idea is to construct all $E_1, E_2, ... E_m$ in a single scan. During the graph search we compute, for each edge *e* being scanned, the *i* satisfying $e \in E_i$. Such *i* can be defined to be the smallest such that if *e* is put in E_i, the condition: each node in any E_j ($j = 1, ..., i$) is visited only once, is not violated, where E_i denotes the edge sets constructed so far. In the algorithm, we always choose an unvisited edge *e* that is adjacent to edge *e'* Î E_i with the largest *i*. In the algorithm, we associate each node *v* with a label $l(v)$: $l(v) = i$ indicates that *v* has been reached by an edge of the forest $T_{max}(G_{i-1}) = (V_i, E_i)$. In the following algorithm, we assume that the nodes are numbered in terms of the depth-first search.

```
Algorithm find-forest
input: $G = (V, E)$
output: $E_1, E_2, ..., E_m$
    begin
        $E_1 := E_2 := ... := E_m := \varnothing$;
        Mark all nodes $v \in V$ and all edges $e \in E$ "unvisited";
        $l(v) := 0$ for all $v \in V$;
        while there exist "unvisited" nodes do
            begin
            choose an "unvisited" node $v \in V$ with the largest $l$ and the smallest
            "depth-first" number;
            for each "unvisited" edge $e$ incident to $v$ do
                begin
                    Let $u$ be the other end node of $e$ ($\neq v$);
*                   $E_{l(u)+1} := E_{l(u)+1} \cup \{e\}$;
**                  $l(u) := l(u) + 1$;
***                 if $l(v) < l(u)$ then $l(v) := l(u) - 1$;
                    Mark $e$ "visited";
                end
            Mark x "visited";
            end
    end
```

For example, by applying the above algorithm to the graph shown in Figure 2(a), we will obtain the edges of three reachable trees shown in Figure 3(b). In the Appendix, we will trace the execution of the algorithm against Figure 3(a) for a better understanding.

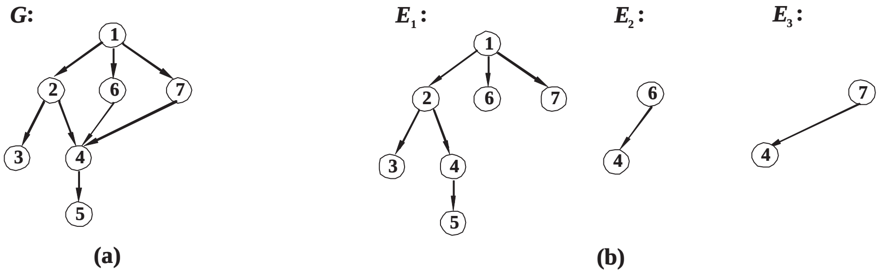
In the above algorithm, each edge is visited exactly once. Therefore, the time complexity of the algorithm is bounded by $O(n + e)$. In the following, we prove a theorem to establish the correctness of the algorithm.

Proposition 2. Applying Algorithm *"find-forest"* to a DAG *G,* a sequence of reachable trees w.r.t. *G* will be found, which covers all of its edges.

Proof. First, we note that by the algorithm, each edge will be visited exactly once and put in some E_i. Therefore, the union of all E_is will contains all edges of *G*. To prove the theorem, we need now to specify that in every E_i, except the root nodes of E_i, each node can be reached along only one path, or say, visited exactly one time w.r.t. E_i. Pay attention to the lines marked with * and **. If a node *u* is visited several times along different edges, such edges will be put in different E_is. Therefore, in each E_i, *u* can be visited only once. By the line marked with ***, if an edge (v, u) is put in some E_i, then an unvisited edge reaching *v* afterwards will be put in E_i or E_{i+1}. If in E_i there is no edge reaching *v* up to now (in this case, $l(v) < l(u)$ holds), the label of *v* will be changed to $i - 1$. Then, if afterwards an unvisited edge reaches *v*, it will be put in E_i. Otherwise, $l(v) = l(u)$ and there must already be an edge in E_i reaching *v*. Thus, if afterwards an unvisited edge reaches *v*, it will be put in E_{i+1}. In this way, in E_i, *v* can be visited only once, which completes the theorem proof.

Now we can label each E_i in the same way as discussed in the previous section. (A forest can be regarded as a tree with a virtual root which has a virtual edge linking each tree root of the forest.) In addition, we notice that a node may appear in several E_is. For example, in Figure 3(b), node 6 appears in E_1 and E_2 while node 4 occurs in all the three reachable trees. Then, after labeling each E_i, each node *v* will get a pair sequence of the form: $(pre_{i_1}, post_{i_1}).(pre_{i_2}, post_{i_2}). \ldots (pre_{i_j}, post_{i_j})$, where for each $i_k \in \{1, ..., m\}$ (*m*

Figure 3. DAG and its Node-Disjunct Maximal Trees



is the in-degree of v) and (pre_{i_k} , $post_{i_k}$) stands for the preorder number and postorder number of v w.r.t. E_{i_k} . In the subsequent discussion, we also say that a label belongs to some E_i, referring to the fact that this pair is calculated in terms of E_i. In terms of such a data structure, we give a naive algorithm below.

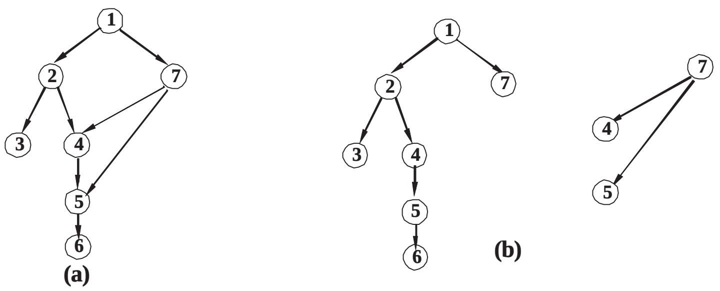
```
$\Delta_{global} := \emptyset$;
$\Delta_{local} := \emptyset$;
$S := \{x\}$;            (* The descendants of $x$ will be searched. *)
**function** *recursion*($S$)
begin
    **for** each $x \in S$ **do** {
        let $p_1.p_2. ... p_m$ be the pair sequence associated with $x$;
        **for** $i = m$ to 1 **do** {
*           let $\Delta$ be the set of descendants of $x$ w.r.t. $E_i$ (evaluated using $p_i$);
**          for each $y \in \Delta$, remove the pair belonging to $E_i$ from the pair sequence
            associated with $y$;
            $\Delta_{local} := \Delta_{local} \cup \Delta$;}}
$\Delta_{local} := \Delta_{local} - \Delta_{global}$;
$\Delta_{global} := \Delta_{global} \cup \Delta_{local}$;
call *recursion*($\Delta_{local}$);
end
```

In the above algorithm, pay attention to the line marked with *, by which all the descendants of x will be evaluated in E_i, using p_i. Since these descendants may appear also in other E_js, they should be used for the further computation. But the pair belonging to E_i has to be eliminated from the pair sequences associated with these nodes to avoid the repeated access to the edges in E_i, which is done by the line marked with **.

The above algorithm suffers, however, from redundancy as discussed.

The graph shown in Figure 4(a) can be decomposed into two reachable trees as shown in Figure 4(b). Applying *recursion*(7) to this graph, the descendant set evaluated in the first **for** loop is {4, 5}. In the second **for** loop, the descendants of nodes 4 and 5 will be computed, which are s_1 = {5, 6} (the descendants of node 4) and s_2 = {6} (the descendants of node 5), respectively. Obviously, s_2 is completely covered by s_1. Therefore, the work of evaluating s_2 can be saved. To this end, we associate each E_i with

Figure 4. Illustration of Redundancy of Recursion (S)



a bit string of size n, denoted B_i. If some node j is a descendant evaluated w.r.t. E_i, the jth bit of B_i will be set to 1, i.e., $B_i[j] = 1$. If the descendants of a node k w.r.t. E_i will be evaluated, we first check $B_i[k]$ to see whether it is equal to 1. If so, the corresponding computation will not be made. Another problem is that if s_2 is evaluated first, the redundant work cannot be avoided even though the checking is performed. Thus, the order of the nodes whose descendants will be evaluated is important. With respect to an E_i, the nodes with smaller preorder numbers will be treated earlier than those with a larger preorder number. This is because a node with a larger preorder number may be a descendant of a node with a smaller one, but not vice versa. In order to sort the nodes in this way, we have to change the control method of the above algorithm. Assume that each node v is associated with a pair sequence of the form: $p_1.p_2. \ldots p_m$, where m is the largest in-degree of the graph. If v does not appear in E_i, p_i will be of the form: (_, _) and will be ignored by sorting. The nodes, whose descendants w.r.t. E_i are going to be evaluated, will first be sorted in terms of p_i. Then, the descendants of these nodes w.r.t. E_i will be computed. In a second loop, the nodes will be sorted again in terms of p_{i-1}. This process repeats until all p_is are handled. Below is the corresponding algorithm with the checking mechanism used.

```
$\Delta_{global} := \emptyset$;
$\Delta_{local} := \emptyset$;
$S := \{x\}$;                (* The descendants of $x$ will be searched. *)
let $p_1.p_2. \ldots p_m$ be the pair sequence associated with each node of the graph;
**for** $i = 1$ to $m$ **do** $B_i = 0$;
**function** *refined-recursion*($S$)
begin
    **for** $i = m$ to 1 **do** {
        sort $S$ in terms of $p_i$s;
        let the sorted $S$ be $\{v_1, \ldots, v_k\}$;
        **for** $j = 1$ to $k$ **do** {
        if $B_i[v_j] = 0$ then $\Delta :=$ the set of descendants of $v_j$ w.r.t. $E_i$ (evaluated using $p_i$);
        **for** each $v_j \in \Delta$ **do** $\{B_i[v_j] := 1\}$
        $\Delta_{local} := \Delta_{local} \cup \Delta$;}}
    $\Delta_{local} := \Delta_{local} - \Delta_{global}$;
    $\Delta_{global} := \Delta_{global} \cup \Delta_{local}$;
    call *refined-recursion*($\Delta_{local}$);
end
```

Note that we take only O(1) time to check a bit in the bit string. For each newly evaluated node set (each time stored in Δ_{local} in the above algorithm), sorting operations will be performed. But each node v in Δ_{local} can take part in the sorting only d times, where d represents the in-degree of v, since for each node v only d pairs in the pair sequence associated with it is not of the form: (_, _). Assume that each time only Δ_{ij} from $\Delta_i (= \Delta_{local})$ participates in the sorting. Then, the total cost for sorting is:

$$\sum_i \sum_j |\Delta_{ij}| \cdot \log |\Delta_{ij}| \leq e \times \log n.$$

Since each edge is visited at most once, the traversal of the graph needs only O(e) time. Therefore, the time complexity of the algorithm is bounded by O($e \times \log n$), a little bit less than the time required by an algorithm using an adjacency list. More importantly, this algorithm is quite suitable for a relational environment. Furthermore, we can store the data in a special way to support the sorting operation so that no extra time is required. For example, we can define two simple relations to accommodate the graph and its pair sequences as follows:

```
node(Node_id, Node_rest),
reachable_forest(E_num, label_pair, Node_id).
```

The first relation stores all the nodes of the graph. The second relation stores all the reachable trees, in which "E_num" is for the identifiers of the reachable trees. If for each of them the label pairs are stored in the increasing order of their preorder numbers, the sorting operations in the algorithm *refined-recursion*() can be removed. Then, the time complexity of the algorithm can be reduced to O(e). This can be done as follows. Whenever some E_i is considered during the execution, we take the tuples with E_num = i from the relation "reachable_forest." Then, we scan these tuples and check, for each tuple, to see whether B_i[node_id] = 1. If it is the case, the corresponding label pair will be put in a list (a temporary data structure) sequentially. Obviously, the list constructed in this way is sorted into the in-creasing order of the preorder numbers w.r.t. E_i.

Recursion W.R.T. Cyclic Graphs

Based on the method discussed in the previous section, we can easily develop an algorithm to compute recursion for cyclic graphs. We can use Tarjan's algorithm for identifying strong connected components (SCCs) to find cycles of a cyclic graph (Tarjan, 1973) (which needs only O($n + e$) time). Then, we think of each SCC as a single node (i.e., condense each SCC to a node). The resulting graph is a DAG. Applying the algorithm *find_forest*( ) to this DAG, we will get a set of forests. For each forest, we can associate each node with a pair as above. Obviously, all nodes in an SCC will be assigned the same pair (or the same pair sequence). For this reason, the method for evaluating the recursion at some node x should be changed. For example, if a graph becomes a tree after condensing each SCC to a node, the select-from-where statements like those given in the third section (against this graph) can be modified as follows. The first query is quite the same as that previously shown:

```
SELECT    label_pair
FROM      Node
WHERE     Node_id = x
```

But the second is changed slightly:

```
SELECT    *
FROM      Node
WHERE     label_pair.preorder ≥ y.preorder
   and    label_pair.postorder ≤ y.postorder
```

By the second query, the nodes in the same SCC as *x* will be regarded as the descendants of *x*.

For general cases, the method for checking ancestor-descendant relationship applies. No extra complexity is caused. Since Tarjan's algorithm runs in O(*n* + *e*) time, computing recursion for a cyclic graph needs only O(*e*) time.

CONCLUSION

In this chapter, a new labeling technique has been proposed. Using this technique, the recursion w.r.t., a tree hierarchy can be evaluated very efficiently. In addition, we have introduced a new algorithm for computing reachable trees, which requires only linear time. Together with the labeling technique, this method enables us to develop an efficient algorithm to compute recursion for directed graphs in O(*e*) time, where *e* represents the number of the edges of the DAG. More importantly, this method is especially suitable for relational databases and much better than the existing methods.

REFERENCES

Abiteboul, S., Cluet, S., Christophides, V., Milo, T., Moerkotte, G. & Simon, J. (1997). Quering documents in object databases. *International Journal of Digital Libraries*, 1(1), 5-19.

Agrawal, A., Dar, S. & Jagadish, H.V. (1990). Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions of Database Systems,* 15(3), 427-458.

Agrawal, R. & Jagadish, H.V. (1989). Materialization and incremental update of path information. *Proceedings of the 5th International Conference on Data Engineering*, Los Angeles, CA, USA, 374-383.

Agarawal, R. & Jagadish, H.V. (1990). Hybrid transitive closure algorithms. *Proceedings of the 16th International VLDB Conference*, Brisbane, Australia, 326-334.

Bancihon, F. & Ramakrishnan, R. (1986). An amateur's introduction to recursive query processing strategies. *Proceedings of the ACM SIGMOD Conference*, Washington, DC, USA, 16-52.

Banerjee, J., Kim, W., Kim, S. & Garza, J.F. (1988). Clustering a DAG for CAD databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(11), 1684-1699.

Carey, M. et al. (1990). An incremental join attachment for Starburst. *Proceedings of the 16th VLDB Conference,* Brisbane, Australia, 662-673.

Cattell, R.G.G. & Skeen, J. (1992). Object operations benchmark. *ACM Transactions on Database Systems*, 17(1), 1-31.

Chen, Y. & Aberer, K. (1998). Layered index structures in document database systems. *Proceedings of the 7th International Conference on Information and Knowledge Management (CIKM)*, Bethesda, MD, USA, 406-413.

Chen, Y. & Aberer, K. (1999). Combining pat-trees and signature files for query evaluation in document databases. *Proceedings of the 10th International DEXA Conference on Database and Expert Systems Application*, Florence, Italy, September. City: Springer Verlag, 473-484.

Dar. S. & Ramarkrishnan, R. (1994). A performance study of transitive closure algorithm. *Proceedings of the SIGMOD International Conference*, Minneapolis, MN, USA, 454-465.

Haskin, R.L. & Lorie, R.A. (1982). On extending the functions of a relational database system. *Proceedings of the ACM SIGMOD Conference*, Orlando, FL, USA, 207-212.

Ioannidis, Y.E., Ramakrishnan R. & Winger, L. (1993). Transitive closure algorithms based on depth-first search. *ACM Transactions on Database Systems*, 18(3), 512-576.

Jagadish, H.V. (1990). A compression technique to materialize transitive closure. *ACM Transactions on Database Systems*, 15(4), 558-598.

Knuth, D.E. (1973). *The Art of Computer Programming: Sorting and Searching*, London: Addison-Wesley.

Mehlhorn, K. (1984). *Graph Algorithms and NP-Completeness*: *Data Structure and Algorithm 2*. Berlin: Springer-Verlag.

Mendelzon, A.O., Mihaila, G.A. & Milo, T. (1997). Querying the World Wide Web. *International Journal of Digital Libraries*, 1(1), 54-67.

Shimon. E. (1979). *Graph Algorithms.* City, MD: Computer Science Press.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal of Computing,* 1(2), 146-140.

Teuhola, J. (1996). Path signatures: A way to speed up recursion in relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(3), 446-454.

Valduriez, P. & Boral, H. (1986). Evaluation of recursive queries using join indices. *Proceedings of the 1st Workshop on Expert Database Systems*, Charleston, SC, USA, 197-208.

APPENDIX

In this appendix, we trace the algorithm *find-reachable-tree* against the tree shown in Figure 3(a).

See Figure 5. At the beginning, every $r(v)$ is set to 0. After the first loop, the l-value of node 1 remains 0. But the l-values of 2, 6 and 7 are changed to 1. Moreover, node 1 and edge (1, 2), (1, 6) and (1, 7) are marked with "v" to indicate that they have been visited. In addition, part of E_1 has been generated. The rest of the steps are listed in Figures 6, 7 and 8.

Figure 5. The First Execution Step of Find-Node-Disjunct-Forest

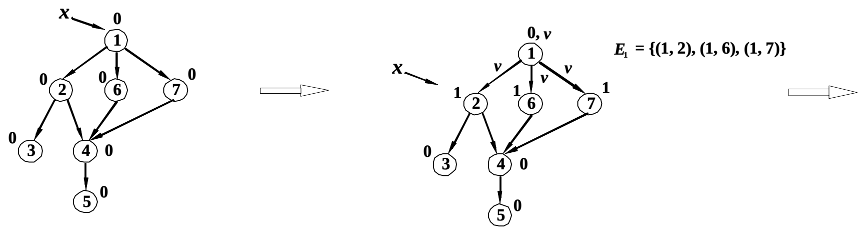


Figure 6. The Second and Third Execution Step of Find-Node-Disjunct-Forest

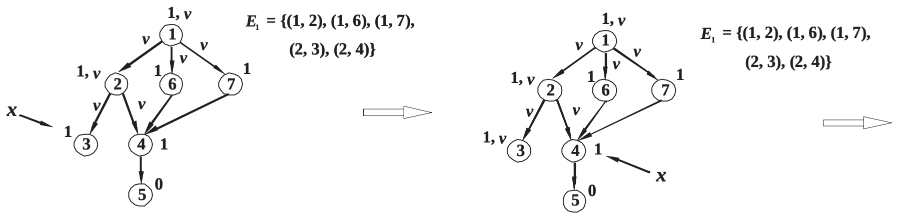

E_1 = {(1, 2), (1, 6), (1, 7), (2, 3), (2, 4)}
E_1 = {(1, 2), (1, 6), (1, 7), (2, 3), (2, 4)}

Figure 7. The Fourth and Fifth Execution Step of Find-Node-Disjunct-Forest

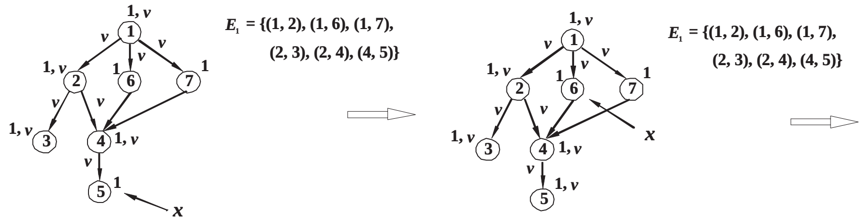
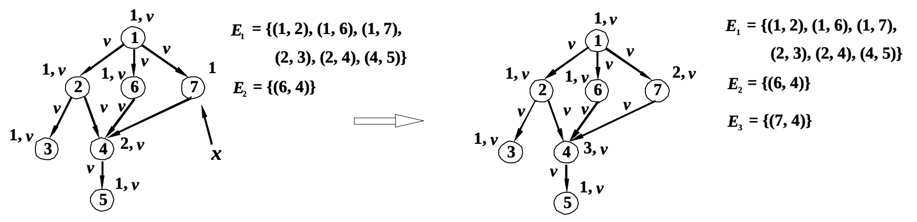

E_1 = {(1, 2), (1, 6), (1, 7), (2, 3), (2, 4), (4, 5)}
E_1 = {(1, 2), (1, 6), (1, 7), (2, 3), (2, 4), (4, 5)}

Figure 8. The Sixth and Seventh Execution Step of Find-Node-Disjunct-Forest


E_1 = {(1, 2), (1, 6), (1, 7), (2, 3), (2, 4), (4, 5)}
E_2 = {(6, 4)}
E_1 = {(1, 2), (1, 6), (1, 7), (2, 3), (2, 4), (4, 5)}
E_2 = {(6, 4)}
E_3 = {(7, 4)}