On the Transitive Closure Representation and Adjustable Compression

Yangjun Chen* and Donovan Cooke
Dept. of Applied Computer Science
University of Winnipeg, Manitoba, Canada R3B 2E9
{ychen2@uwinnipeg.ca, holy_spawn@yahoo.com}

ABSTRACT

A composite object represented as a directed graph (digraph for short) is an important data structure that requires efficient support in CAD/CAM, CASE, office systems, software management, web databases, and document databases. It is cumbersome to handle such objects in relational database systems when they involve ancestor-descendant relationships (or say, recursive relationships). In this paper, we present a new encoding method to label a digraph, which reduces the footprints of all previous strategies. This method is based on a tree labeling method and the concept of branchings that are used in graph theory for finding the shortest connection networks. A branching is a subgraph of a given digraph that is in fact a forest, but covers all the nodes of the graph. On the one hand, the proposed encoding scheme achieves the smallest space requirements among all previously published strategies for recognizing recursive relationships. On the other hand, it leads to a new algorithm for computing transitive closures for DAGs (directed acyclic graph) in O(e·b) time and O(n·b) space, where n represents the number of the nodes of a DAG, e the numbers of the edges, and b the DAG's breadth. The method can also be extended to graphs containing cycles. Especially, based on this encoding method, a multi-level compression is developed, by means of which the space for the representation of a transitive closure can be reduced to $O((b/d^k)\cdot n)$, where *k is the number of compression levels and d is the average outdegree of the nodes.*

Categories & Subject Decriptors: H.2.4
General Terms: Databases**,** Algorithms, Performance
Key Words: directed acyclic graphs, transitive closures, branchings, topological order, graph decomposition

1. INTRODUCTION

It is a general opinion that relational database systems are inadequate for manipulating composite objects that arise in novel applications such as web and document databases [11, 12], CAD/CAM, CASE, office systems and software management [7, 27, 45]. Especially, when recursive relationships are involved, it is cumbersome to handle them in relational database environments, which sets current relational systems far behind the navigational ones [30, 32].

A composite object can be generally represented as a directed graph (digraph). For example, in a CAD database, a composite object corresponds to a complex design, which is composed of several subdesigns [7]. Often, subdesigns are shared by more than one higher-level designs, 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 hierarchical structure.

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 [10]. The so-called *join index* is another auxiliary access path to mitigate this difficulty [48, 49]. Also, several advanced join algorithms have been suggested, based on hashing and a large main memory, see, e.g., [39]. In addition, a different kind of attempts to attain a compromise solution is to extend relational databases with new features, such as *clustering* of composite objects, by which the concatenated foreign keys of ancestor paths are stored in a primary key (see [22, 33, 38] for detailed description). Another extension to relational system is *nested relations* (or NF^2 relations, see, e.g., [16]). Although it can be used to represent composite objects without sacrificing the relational theory, it suffers from the problem that subrelations cannot be shared. Moreover, recursive relationships cannot be represented by simple nesting because the depth is not fixed. Finally, *deductive database*s and *object-relational databases* can be considered as two quite different extensions to handle this problem [13, 28, 36].

In this paper, we discuss a new encoding approach to pack "ancestor paths" in a relational environment. The main idea of this method is *tree labeling*, by means of which each node v is associated 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 DAGs or digraphs containing cycles by decomposing a graph into a series of trees, for which the approach described above can be employed. As we can see later, a new method for computing recursion efficiently in a relational environment can be developed based on these techniques. In fact, it is a new algorithm to handle this problem with a representation of transitive closures different from traditional ones. It needs only $O(e\cdot b)$ time and $O(n\cdot b)$ space, where b is the breadth of the graph, defined to be the least number of disjoined paths that cover all the nodes of a graph. This computational complexity is better than any existing method for this problem, including the graph-based algorithms [18, 19, 34, 35, 37], the graph encoding [1, 2, 6, 14, 26, 45, 50] and the matrix-based algorithms [23, 31, 46, 47]. With such a representation, the time for path checking to see whether a node is a descendant of another is on $(\log_2 b)$. Furthermore, the representation of a transitive closure can be compressed in an adjustable way to fit in different cases. For instance, for a large transitive closure, we can use high level compression while for a small one, we use low level compression or no compression at all. Generally, using a k-level compression, the space overhead can be reduced to $O((b/d^k)\cdot n)$, where d is the average outdegree of the nodes.

The remainder of the paper is organized as follows. Section 2 is on a simple tree labeling. Section 3 elaborates the technique to arbitrary recursion pattern. In Section 4, we discuss how the representation of a transitive closure can be compressed. Section 5 is devoted to the

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

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

maintenance of a compressed transitive closure. In Section 6, we show how this method can be used in a relational database. Finally, a short conclusion is set forth in Section 7.

2. TREE LABELING

In this section, we mainly discuss the concepts of tree labeling, based on which our algorithm is designed. For any directed tree T, we can label it as follows. 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 a similar way, by traversing T in *postorder*, each node v will get another number $post(v)$. These two numbers can be used to characterize the ancestor-descendant relationships of nodes as follows.

Proposition 1 Let v and v' be two nodes of a tree T. Then, v' is a descendant of v iff $pre(v') > pre(v)$ and $post(v') < post(v)$.

Proof. See Exercise 2.3.2-20 in [29]. □

The following example helps for illustration.

Example 1 See the pairs associated with the nodes of the directed tree shown in Fig. 1. The first element of each pair is the preorder number of the corresponding node and the second is its postorder number. Using such labels, the ancestor-descendant relationships of nodes can be easily checked. For instance, 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*. □

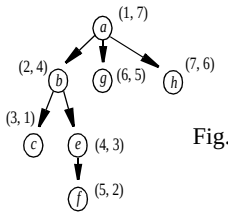


Fig. 1 Labeling a tree

Let (p, q) and (p', q') be two pairs associated with nodes u and v. We say that (p, q) is subsumed by (p', q'), denoted $(p, q) \prec (p', q')$, if $p > p'$ and $q < q'$. Then, u is a descendant of v if (p, q) is subsumed by (p', q').

3. GRAPH DECOMPOSITION AND COMPUTATION OF TRANSITIVE CLOSURES

Now we discuss how to recognize the ancestor-descendant relationships w.r.t. a general structure: a DAG or a graph containing no cycles. First, we address the problem of DAGs in 3.1. Then, cyclic graphs will be discussed in 3.2.

3.1 Recursion w.r.t. DAGs

What we want is to apply the technique discussed above to a DAG. To this end, we establish a *branching* of the DAG as follows.

Definition 1 (*branching* [43]) A subgraph $B = (V, E')$ of a digraph $G = (V, E)$ is called a branching if it is cycle-free and $d_{indegree}(v) \le 1$ for every $v \in V$.

Clearly, if for only one node r, $d_{indegree}(r) = 0$, and for all the rest of the nodes, v, $d_{indegree}(v) = 1$, then the branching is a directed tree with root r. Normally, a branching is a set of directed trees. Now, we assign each edge e a same cost (e.g., let cost $c(e) = 1$ for every edge). We will find a branching for which the sum of the edge costs, $\sum_{e \in E'} c(e)$, is maximum.

For example, the trees shown in Fig. 2(b) are a maximal branching of the graph shown in Fig. 2(a) if each edge has a same cost.

Assume that the maximal branching for $G = (V, E)$ is a set of trees T_i with root r_i ($i = 1, ..., m$). We introduce a *virtual root* r for the branching and an edge $r \rightarrow r_i$ for each T_i, obtaining a tree G_r, called the core of G. For instance, the tree shown in Fig. 2(c) is the core of the graph shown in Fig. 2(a). Using Tarjan's algorithm for finding optimum branchings [43], we can always find a maximal branching for a directed graph in $O(|E|)$ time if the cost for every edge is equal to each other. Therefore, the core tree for a DAG can be constructed in linear time.

By traversing G_r in *preorder*, each node v will obtain a number $pre(v)$; and by traversing G_r in *postorder*, each node v will get another number $post(v)$. These two numbers can be used to recognize the ancestor-descendant relationships of all G_r's nodes as discussed in Section 2.

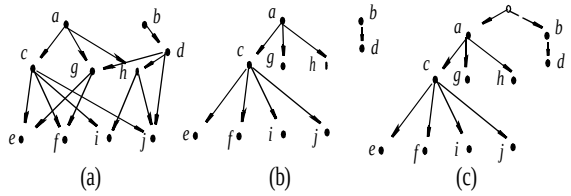


Fig. 2 A DAG and its branching

In a G_r (for some G), a node v can be considered as a representation of the subtree rooted at v, denoted $T_{sub}(v)$; and the pair $(pre, post)$ associated with v can be considered as a pointer to v, and thus to $T_{sub}(v)$. (In practice, we can associate a pointer with such a pair to point to the corresponding node in G_r.) In the following, what we want is to construct a pair sequence: $(pre_1, post_1)$, ..., $(pre_k, post_k)$ for each node v in G, representing the union of the subtrees (in G_r) rooted respectively at $(pre_j, post_j)$ ($j = 1, ..., k$), which contains all the descendants of v. In this way, the space overhead for storing the descendants of a node is dramatically reduced. Later we will shown that a pair sequence contains at most $O(b)$ pairs, where b is the breadth of G.

Example 2 The core tree G_r of the DAG G shown in Fig. 2(a) can be labeled as shown in Fig. 3(a). Then, each of the generated pairs can be considered as a representation of some subtree in G_r. For instance, pair (3, 5) represents the subtree rooted at c in Fig. 3(a).

If we can construct, for each node v, a pair sequence as shown in Fig. 3(b), where it is stored as a linked list, the descendants of the nodes can be represented in an economical way. Let $L = (pre_1, post_1), ..., (pre_k, post_k)$ be a pair sequence and each $(pre_i, post_i)$ be a pair labeling v_i ($i = 1, ..., k$). Then, L corresponds to the union of the subtrees $T_{sub}(v_1)$, ..., and $T_{sub}(v_k)$. For example, the pair sequence (4, 1)(5, 2)(8, 6) associated with g in Fig. 3(b) represents a union of 3 subtrees: $T_{sub}(e)$, $T_{sub}(f)$, and $T_{sub}(g)$, which contains all the descendants of g in G. □

The question is how to construct such a pair sequence for each node v so that it corresponds to a union of subtrees in G_r, which contains all the descendants of v in G.

First, we notice that by labeling G_r, each node in $G = (V, E)$ will be initially associated with a pair as illustrated in Fig. 4. That is, if a node v is labeled with $(pre, post)$ in G_r, it will be initially labeled with the same pair $(pre, post)$ in G.

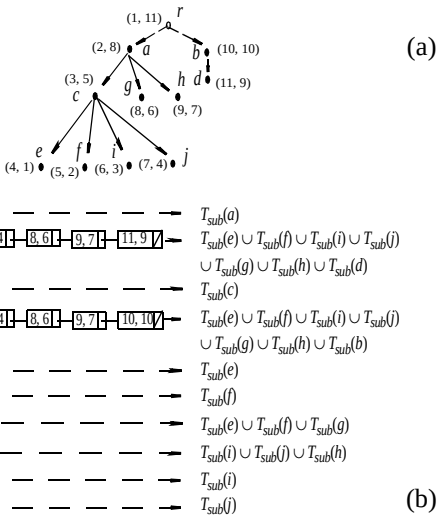


Fig. 3. Tree labeling and illustration for transitive closure representation

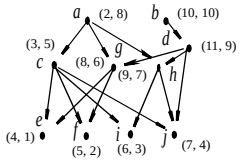


Fig. 4. Graph labeling

To compute the pair sequence for each node, we sort the nodes of G topologically, i.e., $(v_i, v_j) \in E$ implies that v_j appears before v_i in the sequence of the nodes. The pairs to be generated for a node v are simply stored in a linked list A_v. Initially, each A_v contains only one pair produced by labeling G_r.

We scan the topological sequence of the nodes from the beginning to the end and at each step we do the following:

> Let v be the node being considered. Let v_1, ..., v_k be the children of v. Merge A_v with each A_{v_i} for the child node v_i (i = 1, ..., k) as follows. Assume $A_v = p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_g$ and $A_{v_i} = q_1 \rightarrow q_2 \rightarrow ... \rightarrow q_h$, as illustrated in Fig. 5. Assume that both A_v are increasingly ordered. (We say a pair p is larger than another pair p', denoted $p > p$' if $p.pre > p'.pre$ and $p.post > p'.post$.)

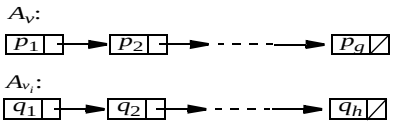


Fig. 5. linked lists associated with nodes in G

We step through both A_v and A_{v_i} from left to right. Let p_i and q_j be the pairs encountered. We'll make the following checkings to merge A_{v_i} into A_v.

(1) If $p_i.pre > q_j.pre$ and $p_i.post > q_j.post$, insert q_j into A_v after p_{i-1} and before p_i and move to q_{j+1}.

(2) If $p_i.pre > q_j.pre$ and $p_i.post < q_j.post$, remove p_i from A_v and move to p_{i+1}. (*p_i is subsumed by q_j.*)

(3) If $p_i.pre < q_j.pre$ and $p_i.post > q_j.post$, ignore q_j and move to q_{j+1}. (*q_j is subsumed by p_i; but it should not be removed from A_{v_i}.*)

(4) If $p_i.pre < q_j.pre$ and $p_i.post < q_j.post$, ignore p_i and move to p_{i+1}.

(5) If $p_i = p_j$' and $q_i = q_j$', ignore both (p_i, q_i) and (p_j', q_j'), and move to (p_{i+1}, q_{i+1}) and (p_{j+1}', q_{j+1}'), respectively.

We notice that initially each A_v contains only one pair and is trivially sorted. Then, when we merge a sorted pair sequence into another sorted pair sequence as above, the result pair sequence must also be sorted.

Proposition 1 The space used to store a transitive closure is bounded by $O(n \cdot b)$, where b is the breadth of G (which is defined to be the least number of disjoined paths that cover all the nodes) and n is the number of the nodes of G.

Proof. In the above procedure, each node v is associated with a linked list A_v. We claim that the size of A_v is bounded by b. Assume that A_v contains $b + 1$ pairs that are different from each other. Then, there must exists two pairs p and q so that p subsumes q or *vice versa*. Therefore, the space needed for Algorithm *all-sequence-generation* is bounded by $O(n \cdot b)$. □

Proposition 2 The time used to generate a transitive closure is bounded by $O(e \cdot b)$, where b is the breadth of G and e is the number of the edges of G.

Proof. Similar to Proposition 1. □

4. COMPRESSING A TRANSITIVE CLOSURE

In this section, we discuss a trade-off between storage space and retrieving time by introducing an adjustable compression method to reduce the size of pair sequences.

4.1 1-level compression

In Fig. 3, we notice that the pair sequences associated with nodes d and b are almost the same, and different only at the last pair. This hints a possible space reduction, but at cost of more retrieving time. In the following, we address this issue in detail. For ease of explanation, we first discuss a *1-level* compression. Then, extend this idea to the so-called *multi-level* compression.

To achieve the 1-level compression, we associate each node v with three sequences: $L_0(v)$, $L_1(v)$, and $L_2(v)$, defined as follows:

$L_0(v)$ - a pair (p, q), where p and q are the preorder and postorder number of v, respectively.

$L_1(v)$ - a pair sequence (pre_1, $post_1$), ..., (pre_k, $post_k$), where each (pre_i, $post_i$) is a pair associated with a child node of v, but not subsumed by (p, q), nor by any pair pointed to by any entry in $L_2(v)$.

$L_2(v)$ - a list of links: l_1, ..., l_j, where each l_i points to a node v_i, whose $L_1(v_i)$ should be merged into $L_1(v)$; but for the compression purpose, $L_1(v_i)$ is replaced with a link stored in $L_2(v)$.

The three sequences for each of the nodes in a graph can be constructed by using the following algorithm.

Algorithm *1-level-compression*
begin
1 Let v_n, v_{n-1}, ..., v_1 be the topological sequence of the nodes of G;
2 **for** i **from** n downto 1 **do**
3 {let i_1, ..., i_l be the child nodes of v_i;
4 **for** j **from** 1 to l **do** {
5 **if** ($L_1(v_{i_j}) \neq nil$) **then**
6 insert a link pointing to v_{i_j} into $L_2(v_i)$;
7 $L_2(v) \leftarrow L_2(v) \cup L_2(v_{i_j})$;
8 **if** ($L_0(v_{i_j})$ not subsumed by $L_0(v)$,
not subsumed by any pair in $L_1(v)$, and
not subsumed by any pair pointed to by any entry in $L_2(v)$)
9 **then** merge $L_0(v_{i_j})$ into $L_1(v)$;}
10 }
end

Example. 3 Applying Algorithm *1-level-compression* to the graph shown in Fig. 2(a), we will produce a data structure as shown in Fig. 6.

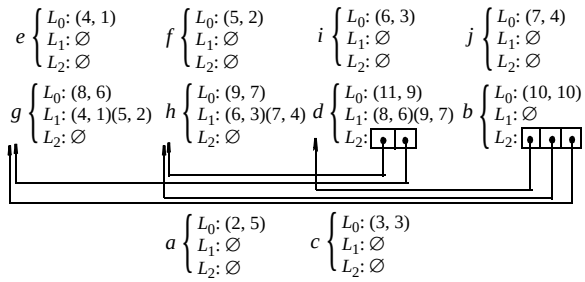


Fig. 6. Illustartion of 1-level compression

From Fig. 6, we can see that $L_2(d)$ contains two links to h and g, respectively. The first of them stands for $L_1(h)$ and the second for $L_1(g)$. In $L_2(b)$, we have three links which represents $L_1(d)$, $L_1(h)$, and $L_1(g)$, respectively. □

With such a data structure, the space is reduced to $O((b/d) \cdot n)$. However, more time is needed to check whether a node u is a descendant

of another node *v*. This can be done in three steps:

(1) Check whether $L_0(u)$ is subsumed by $L_0(v)$;

(2) Check whether $L_0(u)$ is subsumed by any pair in $L_1(v)$;

(3) Check whether $L_0(u)$ is subsumed by any pair pointed to by any entry in $L_2(v)$.

Obviously, the time complexity of this operation is worse than $\log_2 b$, but bounded by $O((1 + b/d)\cdot\log_2 d)$, where d is the average out-degree of the nodes in G and $\log_2 d$ is the measurement of the time for checking a pair for a node u against the pair sequence $L_1(v)$ for another node v.

4.2 Multi-level compression

The idea discussed above can be extended to k-level compression ($k \geq 1$), by which each node is associated with k+2 sequences, defined as follows:

$L_0(v)$ - a pair (p, q), where p and q are the preorder and postorder number of v, respectively.

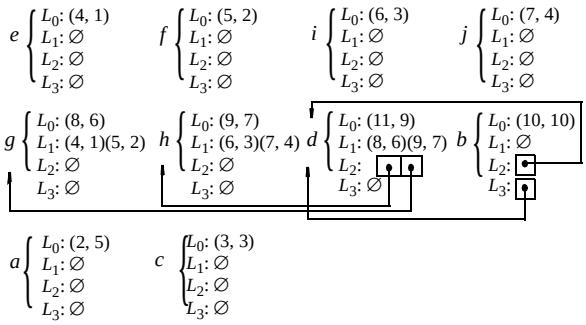
$L_1(v)$ - a pair sequence (pre_1, $post_1$), ..., (pre_h, $post_h$), where each (pre_i, $post_i$) is a pair associated with a child node of v, but not subsumed by (p, q), nor by any pair pointed to by any entry in any $L_l(v)$ ($2 \leq l \leq k$+2).

For each l ($2 \leq l \leq k$+2), $L_l(v)$ is a list of links: l_1, ..., l_j, where each l_i points to a node v_i, whose $L_{l-1}(v_i)$ should be merged into $L_l(v)$; but for the compression purpose, $L_{l-1}(v_i)$ is replaced with a link stored in $L_{l+1}(v)$.

The following is the algorithm to generate a multi-level-compressed representation of a transitive closure.

Algorithm *multi-level-compression*(k) (*k - level of compression*)
begin
1 Let $v_n, v_{n-1}, ..., v_1$ be the topological sequence of the nodes of G;
2 **for** i **from** n downto 1 **do**
3 {let $v_{i_1}, ..., v_{i_l}$ be the child nodes of v_i;
4 **for** j **from** 1 to l **do** {
5 **for** c **from** 2 to k+2 **do**
6 { **if** ($L_{c-1}(v_{i_j}) \neq nil$) **then**
7 insert a link pointing to v_{i_j} into $L_c(v_i)$;}}
8. $L_{k+2}(v) \leftarrow L_{k+2}(v) \cup L_{k+2}(v_{i_j})$;
9 **if** ($L_0(v_{i_j})$ not subsumed by $L_0(v)$,
not subsumed by any pair in $L_1(v)$, and
not subsumed by any pair pointed to by an entry in any $L_j(v)$ ($2 \leq j \leq k$+2)
10 **then** merge $L_0(v_{i_j})$ into $L_1(v)$;}
11 }
end

Example 4 Applying Algorithm *multi-level-compression*(2) to the graph shown in Fig. 2(a), we will produce a data structure as shown in Fig.7.



In Fig. 10, $L_3(b)$ contains a link to d, which represents $L_2(d)$. Along with the links in $L_2(d)$, we can find $L_1(h)$ and $L_1(g)$. □

Using such a data structure to represent the transitive closure of a graph G, the space overhead is bounded by $O((b/d_k)\cdot n)$.

As with the 1-level compression, we check whether a node u is a descendant of another node v in three steps:

(1) Check whether $L_0(u)$ is subsumed by $L_0(v)$;

(2) Check whether $L_0(u)$ is subsumed by any pair in $L_1(v)$;

(3) For each l ($2 \leq l \leq k$+2), do the following:

procedure *label-checking*(l, v)

$c \leftarrow l$;

if $c = 1$ **then** check whether $L_0(u)$ is subsumed by any pair in $L_1(v)$;

else {let $c_1, ..., c_j$ be the nodes pointed to by the entries in $L_c(v)$; for each c_i call label-checking(l - 1, v_{c_i});}

From the above, we can see that in order to check whether a node u is a descendant of another node v, we need to check all the entries in all the sequences of v, along which $O(d^{k-1})$ nodes will be accessed. Therefore, the total time for this task is bounded by $O(d^{k-1} + \frac{b}{d}\cdot\log_2 d)$.

Consider the function

$$f(k) = d^{k-1} + \frac{b}{d}\cdot\log_2 d + (b/d_k)\cdot n.$$

The first two items in $f(k)$ represent the time for path checking while the third item represents the space overhead. It can be shown that when $k = \frac{1}{2} + \log_d\sqrt{bn}$, $f(k)$ reaches its minimum:

$$2\sqrt{\frac{b}{d}\cdot n} + \frac{b}{d}\cdot\log_2 d\,.$$

So we can choose $k = \frac{1}{2} + \log_d\sqrt{bn}$ to make a compression to get a combined optimization for both retrieving time and storage space.

5. MAINTENANCE

In this section, we discuss the maintenance of k-level compression ($k \geq 1$). Mainly, we consider two operations: inserting an edge and inserting a node.

- Inserting an edge

Let v and u be two nodes in G. The insertion of the edge (v, u) can be done as follows.

(1)Insert a link pointing to u to each $L_i(v)$ if $L_{i-1}(u) \neq nil$) ($2 \leq i \leq k$+2);

(2)$L_{k+2}(v) \leftarrow L_{k+2}(v) \cup L_{k+2}(u)$;

(3)**if** ($L_0(u)$ not subsumed by $L_0(v)$,
not subsumed by any pair in $L_1(v)$, and
not subsumed by any pair pointed to by an entry in any $L_j(v)$ ($2 \leq j \leq l$+2))
then merge $L_0(u)$ into $L_1(v)$;}

(4)For any ancestor w of v, $L_{k+2}(w) \leftarrow L_{k+2}(w) \cup L_{k+2}(u)$.

Note that the union operation takes linear time. So the time cost for inserting an edge is on $O((b/d_k)\cdot n)$ since the whole size of all the sequences associated with a node is bounded by $O(b/d_k)$. For the practice purpose, however, each time when we do $L_{k+2}(w) \leftarrow L_{k+2}(w) \cup L_{k+2}(u)$, we can simply establish a link to connect $L_{k+2}(w)$ and $L_{k+2}(u)$, and delay the union op-

eration after many edges are inserted or to a late time point when the system is not busy. In this way, the actual time for inserting an edge is on $(k \cdot n)$.

- Inserting an node

To facilitate the insertion of nodes, we leave a certain gap between each two consecutive preorder (also postorder) numbers. When a new node is inserted, its preorder and postorder numbers are chosen from the corresponding gap as discussed below.

We first consider the calculation of the numbers for a new node when it is inserted into a tree. We distinguish among four cases.

1. A new node *v* is inserted into a tree *T* as a direct right sibling of some node *u*.
2. A new node *v* is inserted into a tree *T* as a parent of some node *u*.
3. A new node *v* is inserted into a tree *T* as a direct left sibling of some node *u*.
4. A new node *v* is inserted into a tree *T* as a child of some node *u* and the parent of one of *u*'s children.

In the first two cases, the pair (*pre*, *post*) associated with *v* is calculated as follows.

Let the pair associated with *u* be (p, q). Let the pair associated with the node *s* preceding *u* (according to the preorder numbering) be (p_s, q_s). Let the pair associated with the node *t* next to *s* (according to the postorder numbering) be (p_t, q_t). We have

$$pre = p_s + \left\lceil \frac{p - p_s}{2} \right\rceil, \text{ and } post = q_t - \left\lfloor \frac{q_t - q}{2} \right\rfloor.$$

Obviously, if each node keeps a pointer to its predecessor (according to the preorder numbering) and a pointer to its successor (according to the postorder numbering), this operation needs only a constant time.

Now we consider another two cases (3) and (4) that are dual to case 1 and 2.

Let the pair associated with *u* be (p, q). Let the pair associated with the node *s* preceding *u* (according to the postorder numbering) be (p_s, q_s). Let the pair associated with the node *t* next to *s* (according to the preorder numbering) be (p_t, q_t). We have

$$pre = p + \left\lceil \frac{p_t - p}{2} \right\rceil, \text{ and } post = q - \left\lfloor \frac{q - q_s}{2} \right\rfloor.$$

To calculate the pair for a new node to be inserted into a DAG, we have to determine where to insert the node in the corresponding branching. This can be done by checking the parent of the new node, which is a easy task since the information on its parent must be specified. Two cases will be considered:

(i) The node *v* is inserted as a child of some node in the branching.

(ii) The node *v* is inserted between an edge (a, b) (i.e., it is inserted as a child of *a* and the parent of *b*.)

For the first case, we will call the algorithm for inserting an edge. For the second case, we also call the algorithm for inserting an edge, but two times: one is for inserting (v, b) and the other is for inserting (a, v). Since the time for determining the pair for a new node is O(1), the insertion of nodes has the same time complexity as the insertion of edges.

6. COMPUTING RECURSION IN RELATIONAL DATABASES

The algorithm discussed in Section 3 hints a new way to speed-up recursion in a relational database.

We can physically store the label pair for each node, as well as its label pair sequence produced using Algorithm *all-sequence-generation.* Concretely, the relational schema to handle recursion w.r.t. a DAG can be established as follows:

Node(Node_id, *label*, *label_sequence*, ...),

where *label* and *label_sequence* are used to accommodate the label pair and the label pair sequence associated with the nodes of a graph, respectively. Then, to retrieve the descendants of node *x*, we issue two queries. The first query is of the following form:

Q_1: SELECT label_sequence
FROM Node
WHERE Node_id = *x*

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

Q_2: SELECT *
FROM Node
WHERE ϕ(label, *y*),

where $\phi(p, s)$ is a boolean function with the input: *p* and *s*, where *p* is a pair and *s* a pair sequence. If there exists a pair *p*' in *s* such that $p \prec p'$ (i.e., $p.pre > p'.pre$ and $p.post < p'.post$), then $\phi(p, s)$ returns *true*; otherwise *false*.

To compute recursion w.r.t. a graph containing cycles, we first issue a query same as Q_1, and then issue another one slightly different from Q_2:

Q_2': SELECT *
FROM Node
WHERE γ(label, *y*),

where $\gamma(p, s)$ is a boolean function with the input: *p* and *s*, where *t* is a pair and *s* a pair sequence. If there exists a pair *p*' in *s* such that $p'.pre \le p.pre$ and $p'.post \ge p.post$, then $\gamma(p, s)$ returns *true*; otherwise *false*. (Note the difference between $\gamma(\ )$ and $\phi(\ )$. In Q_2', any two nodes in the same SCC are considered to be the descendants of each other.)

For a transitive closure which is *k*-level compressed, we need to change the data structure slightly. For each node *v*, we store its sequences in a separate file, and put *l* $(1 \le l \le k)$ addresses in the table Node as a value of the attribute *label_sequence* with the *i*th address pointing to the position where $L_i(v)$ can be found. Accordingly, the functions $\phi(\ )$ and $\gamma(\ )$ should be changed in terms of the discussion in 4.2.

7. CONCLUSION

In this paper, a new technique for labeling a digraph has been proposed. Using this technique, the recursion w.r.t. a tree hierarchy can be computed very efficiently. In addition, we have devised an algorithm to generate pair sequences for the nodes of a digraph which can be used to identify the ancestor-descendant relationship of the nodes. The method needs $O(e \cdot b)$ time and $O(n \cdot b)$ space, where *e* is the number of the edges of a digraph, *n* is the number of the nodes, and *b* is its breadth. More importantly, a multi-level compression can be conducted based on the graph encoding method and is adjustable for different cases. For an *k*-level compression $(k \ge 1)$, the space complexity is on $O((b/d_k) \cdot n)$, where *d* is the average outdegree of the nodes.

REFERENCES

[1] S. Abdeddaim, On Incremental Computation of Transitive Closure and Greedy Alignment, in: *Proc. 8th Symp. Combinatorial Pattern Matching*, ed. Alberto Apostolico and Jotun Hein, 1997, pp. 167-179.

[2] R. Agrawal, A. Borgida and H.V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," *Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data*, Oregon, 1989, pp. 253-262.

[3] R. Agrawal, S. Dar, H.V. Jagadish, "Direct transitive closure algorithms: Design and performance evaluation," *ACM Trans. Database Syst.* 15, 3 (Sept. 1990), pp. 427 - 458.

[4] R. Agrawal and H.V. Jagadish, "Materialization and Incremental Update of Path Information," in: *Proc. 5th Int. Conf. Data*

Engineering, Los Angeles, 1989, pp. 374 - 383.

[5] R. Agarawal and H.V. Jagadish, "Hybrid transitive closure algorithms," In *Proc. of the 16th Int. VLDB Conf.*, Brisbane, Australia, Aug. 1990, pp. 326 -334.

[6] M.F. van Bommel and T.J. Beck, "Incremental Encoding of Multiple Inheritance Hierarchies Supporting Lattice Operations, *Linkoping Electronic Articles in Computer and Information Science*, http://www.ep.liu.se/ea/cis/2000/001.

[7] J. Banerjee, W. Kim, S. Kim and J.F. Garza, "Clustering a DAG for CAD Databases," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 14, No. 11, Nov. 1988, pp. 1684 - 1699.

[8] K.S. Booth and G.S. Leuker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *J. Comput. Sys. Sci*., 13(3):335-379, Dec. 1976.

[9] F. Bancihon and R. Ramakrishnan, "An Amateurs Introduction to Recursive Query Processing Strategies," in: *Proc. ACM SIGMOD Conf.*, Washington D.C., 1986, pp. 16 - 52.

[10] M. Carey et al., "An Incremental Join Attachment for Starburst," in: *Proc. 16th VLDB Conf.*, Brisbane, Australia, 1990, pp. 662 - 673.

[11] Y. Chen, K. Aberer, "Layered Index Structures in Document Database Systems," *Proc.* 7th *Int. Conference on Information and Knowledge Management (CIKM)*, Bethesda, MD, USA: ACM, 1998, pp. 406 - 413.

[12] Y. Chen and K. Aberer, "Combining Pat-Trees and Signature Files for Query Evaluation in Document Databases," in: *Proc. of 10th Int. DEXA Conf. on Database and Expert Systems Application*, Florence, Italy: Springer Verlag, Sept. 1999. pp. 473 - 484.

[13] Y. Chen, "On the Graph Traversal and Linear Binary-chain Programs," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, No. 3, May 2003, pp. 573-596.

[14] N.H. Cohen, "Type-extension tests can be performed in constant time," *ACM Transactions on Programming Languages and Systems*, 13:626-629, 1991.

[15] R.G.G. Cattell and J. Skeen, "Object Operations Benchmark," *ACM Trans. Database Systems*, Vol. 17, no. 1, pp. 1 -31, 1992.

[16] P. Dadam et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies," *Proc. ACM SIGMOD Conf*., Washington D.C., 1986, pp. 356-367.

[17] S. Dar and R. Ramarkrishnan, "A Performance Study of Transitive Closure Algorithm," in *Proc. of SIGMOD Int. Conf.*, Minneapolis, Minnesota, USA, 1994, pp. 454 - 465.

[18] J. Dzikiewicz, "An Algorithm for Finding the Transitive Closure of a Digraph," *Computing* 15, 75 - 79, 1975.

[19] J. Ebert, "A Sensitive Transitive closure Algorithm," *Inf. Process Letters* 12, 5 (1981).

[20] J. Eve and R. Kurki-Suonio, "On Computing the Transitive Closure of a Relation," *Acta Informatica* 8, 303 - 314, 1977.

[21] M. Fredman and R. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *proc. 25th IEEE Symp. on Foundations of Computer Science*, pp. 338-346, 1984.

[22] R.L. Haskin and R.A. Lorie, "On Extending the Functions of a Relational Database System," *Proc. ACM SIGMOD Conf.*, Orlando, Fla., 1982, pp. 207-212.

[23] T. Ibaraki and N. Katoh, On-line Computation of transitive closure for graphs, *Information Processing Letters*, 16:95-97, 1983.

[24] G.F. Italiano, Amortized efficiency of a path retrieval data structure, *Theoretical Computer Science*, 48:273-281, 1986.

[25] Y.E. Ioannidis, R. Ramakrishnan and L. Winger, "Transitive Closure Algorithms Based on Depth-First Search," *ACM Trans. Database Syst.*, Vol. 18. No. 3, 1993, pp. 512 - 576.

[26] H.V. Jagadish, "A Compression Technique to Materialize Transitive Closure," *ACM Trans. Database Systems*, Vol. 15, No. 4, 1990, pp. 558 - 598.

[27] T. Keller, G. Graefe and D. Maier, "Efficient Assembly of Complex Objects," *Proc. ACM SIGMOD conf.* Denver, Colo., 1991, pp. 148-157.

[28] W. Kim, "Object-Oriented Database Systems: Promises, Reality, and Future," *Proc. 19th VLDB conf.*, Dublin, Ireland, 1993, pp. 676-687.

[29] D.E. Knuth, *The Art of Computer Programming, Vol.1*, Addison-Wesley, Reading, 1969.

[30] H.A. Kuno and E.A. Rundensteiner, "Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10. No. 5, 1998, pp. 768-792.

[31] J.A. La Poutre and J. van Leeuwen, Maintenance of Transitive closure and transitive reduction of graphs, in *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 106-120. *Lecture Notes in Computer Science 314*, Springer-Verlag, 1988.

[32] W.C. Lee and D.L Lee, "Path Dictionary: A New Access Method for Query Processing in Object-Oriented Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10. No. 3, 1998, pp. 371-388.

[33] B. Lindsay, J. McPherson and H. Pirahesh, "A Data Management Extension Architecture," *Proc. ACM SIGMOD conf*., 1987, pp. 220-226.

[34] K. Mehlhorn, "*Graph Algorithms and NP-Completeness*: *Data Structure and Algorithm 2*" Springer-Verlag, Berlin, 1984.

[35] P. Purdom, "A Transitive Closure Algorithm," *BIT* 10, 76 - 94, 1970.

[36] R. Ramakrishnan and J.D. Ullman, "A Survey of Research in Deductive Database Systems," *J. Logic Programming*, May, 1995, pp. 125-149.

[37] L. Schmitz, "An Improved Transitive Closure Algorithm," *Computing* 30, 359 - 371 (1983).

[38] M. Stonebraker, L. Rowe and M. Hirohama, "The Implementation of POSTGRES," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 1, 1990, pp. 125-142.

[39] L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," ACM Trans. Database Systems, vol. 11, no. 3, 1986, pp. 239-264.

[40] M.A. Schubert and J. Taugher, "Determing type, part, colour, and time relationship," 16 (special issue on Knowledge Representation):53-60, Oct. 1983.

[41] D.D. Sleator and R. Tarjan, Amortized efficiency of list update rules, *Proc. 16th ACM Symp. on Theory of Computing*, pp.488-492, 1984.

[42] R. Tarjan: Depth-first Search and Linear Graph Algorithms, *SIAM J. Compt*. Vol. 1. No. 2. June 1972, pp. 146 -140.

[43] R. Tarjan: Finding Optimum Branching, *Networks*, 7. 1977, pp. 25 -35.

[44] R. Tarjan, Amortized computational complexity, *SIAM J. Algebraic Discrete Methods 6*, pp. 306-318, 1985.

[45] J. Teuhola, "Path Signatures: A Way to Speed up Recursion in Relational Databases," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 8, No. 3, June 1996, pp. 446 - 454.

[46] S. Warshall, "A Theorem on Boolean Matrices," *JACM*, 9. 1(Jan. 1962), 11 - 12.

[47] H.S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Commun. ACM* 18, 4 (April 1975), 218 - 220.

[48] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices," in: *Proc. 1st Workshop on Expert Database Systems*, Charleston, S.C., 1986, pp. 197 - 208.

[49] P. Valduriez, S. Khoshafian and G. Copeland, "Implementation Techniques of Complex Objects," *Proc. 12th VLDB Conf*., Kyoto, Japan, 1986, pp. 101-109.

[50] Y. Zibin and J. Gil, "Efficient Subtyping Tests with PQ-Encoding," *Proc. of the 2001 ACM SIGPLAN conf. on Object-Oriented Programming Systems, Languages and Application*, Florida, October 14-18, 2001, pp. 96-107.