1. Background

A composite object represented as a directed graph is an important data structure. It is hard to handle such objects in relational database systems when they involve ancestor-descendant relationships. A new encoding method based on a tree labeling method and the concept of branchings was introduced to solve this problem by Dr. Yang jun Chen.

2. Introduction

In a relational system, composite objects must be fragmented across many relations, requiring joins to gather all the parts. A typical approach to improve join efficiency is to equip relations with hidden pointer fields for coupling the tuples to be joined [1]. The other method 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 primary key [2,3]. Besides these methods, the method of labeling a graph has been used to materialize transitive closures. In the past several decades, the computation of transitive closures was an interesting research topic in the graph theory. Dr. Chen combines the method in [4] and the method in [5] to introduce a new method, tree labeling, a new algorithm for computing transitive closures with a representation of transitive closures different from traditional ones. The basic idea of tree labeling is: in a tree hierarchy, we associate each node \( v \) with a pair of integers \( (a, B) \), such that if \( v' \), another node associated with \( (a', B') \), is a descendant of \( v \), some arithmetical relationship between \( a \) and \( a' \), as well as \( B \) and \( B' \) can be determined. Then, such relationships can be used to find all descendants of a node, and the recursive closure can be computed very efficiently.

3. Tree labeling and recursion computation

The main part of Dr. Chen's methods is tree labeling, which not only provides an efficient way to handle recursion in a relational environment, but also leads to a graph algorithm to compute transitive closures with optimal computational complexity.

**Tree labeling:** Consider a tree \( T \), by traversing \( T \) in preorder, each node \( v \) will get a value \( \text{pre}(v) \) 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 value \( \text{post}(v) \). These two numbers can be used to characterize the ancestor-descendant relationships.

**Proposition 1:** Let \( v \) and \( v' \) be two nodes of a tree \( T \). Then, \( v' \) is a descendant of \( v \) if \( \text{pre}(v') > \text{pre}(v) \) and \( \text{post}(v') < \text{post}(v) \) [6].

**Label pair subsumption:** 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 \( (p, q) \subseteq (p', q') \), if \( p > p' \) and \( q < q' \). Then, \( u \) is a descendant of \( v \) if \( (p, q) \) is subsumed by \( (p', q') \).

According to the tree labeling discussed above, the relational schema to handle recursion can consist of only one relation of the following form:

\[
\text{Node}( \text{Node_id}, \text{label_pair}, \text{Node_rest}),
\]
where label_pair is used to accommodate the preorder and the postorder numbers of the nodes of a graph, denoted label_pair.preorder and label_pair.postorder. Then, to retrieve the descendants of node x, we issue two queries as below:

Q1: 
    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 is of the following form:

Q2: 
    SELECT * 
    FROM Node 
    WHERE label_pair.preorder > y.preorder 
    and label_pair.postorder < y.postorder

4. Recursion w.r.t. DAGs

For any DAG, we can establish a branching of the DAG as follows:

**Definition 2.** A subgraph B=(V, E') of a digraph G=(V, E) is called a branching if it is cycle-free and \( d_{\text{indegree}}(v) \leq 1 \) for every \( v \in V \).

If for only one node r, \( d_{\text{indegree}}(v) = 0 \), and for all the rest of the nodes, \( d_{\text{indegree}}(v) = 1 \), then the branching is a directed tree with root r. Furthermore, if we assign every edge e a same cost (i.e., let cost \( c(e)=1 \) for every edge), we will find a branching for which the sum of the edge costs, \( \sum c(e) \), is maximum.

In a \( G_r \), a node v can be considered as a representation of the subtree rooted at v, denoted \( T_{\text{sub}(v)} \); and the pair (pre, post) associated with v can be considered as a pointer to v, and thus to \( T_{\text{sub}(v)} \). The key is how to construct 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. To construct such a sequence, first, we notice that by labeling \( G_r \), each node in G = (V, E) will be initially associated with a pair. 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.

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 link 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:

Let v be the node being considered. Let \( v_1, \ldots, v_k \) 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 \( A_v = p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_k \) and \( A_{v_i} = q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_h \). Assume that both \( A_v \) and \( A_{v_i} \) are increasingly ordered.

We step through both \( A_v \) and \( A_{v_i} \) from left to right. Let \( p_i \) and \( q_i \) be the pairs encountered. We'll make the following checking.

1. If \( p_i.\text{pre} > q_i.\text{pre} \) and \( p_i.\text{post} > q_i.\text{post} \), insert \( q_i \) into \( A_v \) after \( p_{i-1} \) and before \( p_j \) and move to \( q_{i+1} \).

2. If \( p_i.\text{pre} > q_i.\text{pre} \) and \( p_i.\text{post} < q_i.\text{post} \), 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 \( (p_{j+1}', q_{j+1}') \).

Based on the above discussion, we have the following algorithm to merge two pair sequences together.

**Algorithm** pair-sequence-merge( \( A_1, A_2 \))

**Input:** \( A_1 \) and \( A_2 \) - two link lists associated with \( v_1 \) and \( v_2 \).

**Output:** A - modified \( A_1 \), obtained by merging \( A_2 \) into \( A_1 \), containing all the pairs in \( A_1 \) and \( A_2 \) with all the subsumed pairs removed.

**Begin**

1. \( p <- \text{first-element}(A_1) \);
2. \( q <- \text{first-element}(A_2) \);
3. while \( p \neq \text{nil} \) do{
   4.   while \( q \neq \text{nil} \) do{
      5.     if( \( p.\ pre > q.\ pre \) \&\& \( p.\ post < q.\ post \) ) then
         6.         {insert \( q \) into \( A_1 \) before \( p \);
         7.         \( q <- \text{next}(q) \) ;
      8.     else if( \( p.\ pre < q.\ pre \) \&\& \( p.\ post < q.\ post \) ) then
         9.         \( p <- q ;
   10.        \) remove \( p \) from \( A_1 \) ;
   11.        \( p <- \text{next}(p) ;
   12.    \) else if( \( p.\ pre < q.\ pre \) \&\& \( p.\ post < q.\ post \) ) then
         13.        \( p <- \text{next}(p) ;
   14.    \) else if( \( p.\ pre < q.\ pre \) \&\& \( p.\ post < q.\ post \) ) then
         15.        \( p <- \text{next}(p) ;
   16.    \) else if( \( p.\ pre = q.\ pre \) \&\& \( p.\ post = q.\ post \) )
         17.        \( p <- \text{next}(p) ; q <- \text{next}(q) ;
   18.    \) if \( p = \text{nil} \) \&\& \( q = \text{nil} \) then{attach the rest of \( A_2 \) to the end of \( A_1 \) ;}
   19.  end
   20. end

There are some propositions used to clarify the properties of the above algorithm.

**Proposition 2.** Let \( A_1 \) and \( A_2 \) be two pair sequences sorted in increasing order. Let \( A \) be the result obtained by merging \( A_2 \) into \( A_1 \) using Algorithm pair-sequence-merge(). Then, \( A \) is also sorted increasing.

**Proposition 3.** Let \( A_1 \) and \( A_2 \) be two pair sequences sorted in increasing order. Let \( A \) be the result obtained by merging \( A_2 \) into \( A_1 \) using Algorithm pair-sequence-merge(). If \( v \) is a node in a subtree of \( G_{v_i} \), which is rooted at some node labeled with a pair in \( A_2 \), then there must be a pair in \( A \) such that the subtree rooted at it contains \( v \).
Proposition 4. Let $A_1$ and $A_2$ be two pair sequences sorted in increasing order. Let $A$ be the result obtained by merging $A_2$ into $A_1$ using Algorithm pair-sequence-merge(). If $v$ is a node in a subtree of $G_r$, which is rooted at some node labeled with a pair in $A_1$, then there must be a pair in $A$ such that the subtree rooted at it contains $v$.

Proposition 5. The time complexity of Algorithm pair-sequence-merge is bounded by $O(\max\{ |A_1|, |A_2| \})$.

Based on the merging operation discussed above, the pair sequences for all the nodes in a DAG can be computed as follows.

Algorithm all-sequence-generation

\begin{algorithm}
  \begin{algorithmic}
    \Statex 1 let $v_n, v_{n-1}, ..., v_1$ be the topological sequence of the nodes of $G$;
    \Statex 2 for $i$ from $n$ down to 1 do
    \Statex 3 \hspace{1em} \{ let $v_i, ..., v_j$ be the child nodes of $v_i$;
    \Statex 4 \hspace{2em} for $j$ from 1 to $k$ do
    \Statex 5 \hspace{3em} call pair-sequence-merge($A_i, A_j$);
    \Statex 6 \}
  \end{algorithmic}
\end{algorithm}

Proposition 6. The space complexity of Algorithm all-sequence-generation is bounded by $O(nb)$, where $b$ is the breath of $G$ and $n$ is the number of the nodes of $G$.

Proposition 7. The time complexity of Algorithm all-sequence-generation is bounded by $O(eb)$, where $b$ is the breath of $G$ and $e$ is the number of the edges of $G$.

Proposition 8. Let $v$ be a node in $G$. Any descendant $u$ of $v$ must be in a subtree of $G_r$ rooted at a node labeled with a pair in $A$, constructed by Algorithm all-sequence-generation.

The label pair for each node, as well as its label pair sequence produced using Algorithm all-sequence-generation can be stored physically. The relational schema to handle recursion w.r.t. a DAG can be established in the following form:

- Node( Node_id, label, label_sequence, Node_rest),

where label and label_sequence are used to accommodate the label and label pair sequence associated with the nodes of a graph, respectively. Then, to retrieve the descendants of node $x$, we issue two queries.

Q3: \begin{verbatim}
SELECT label_sequence
FROM Node
WHERE Node_id=x
\end{verbatim} 

Let the label sequence obtained by evaluating the above query by $y$:
Q4:  SELECT * 
    FROM Node
    WHERE φ(label,y)
where φ(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 ⊆ p' (i.e., p.pre > p'.pre and p.post < p'.post), then φ(p,s) returns true; otherwise false.

5. Conclusion
In Dr. Chen's research, a new labeling technique has been proposed. With this technique, the recursion w.r.t. a tree hierarchy can be computed without join operations. Using this method, the signature conflicts which may exist using Teuhola's encoding can be avoid. This method can also be extended to compute the recursion w.r.t. DAGs and graphs containing cycles. Given a DAG, a maximal branching can be find as its representation, which can always be labeled. To find the recursive closure of the DAG, we construct a sequence of label pairs for each node, which can be done in a topological order of the DAG in O(e*b) time and O(n*b) space, where n represents the number of the nodes of the DAG, e, the numbers of the edges, and b, the DAG's breath. To compute the recursion w.r.t. a cyclic graph, we first use Tarjan's algorithm to find all its SCCs; and then condense each SCC onto a single node, resulting in a DAG. Finally, we notice that the sequences of label pairs are in fact a new representation of the transitive closure of a graph with a much less space overhead than any existing method.

References