1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

AN EFFICIENT ALGORITHM FOR COMPUTING PARTIAL TRANSITIVE CLOUSERES

Yan Feng *Wen-Chi Hou*

Department of Computer Science
Southern Illinois University at Carbondale
Carbondale, IL 62901 U.S.A hou@cs.siu.edu

Abstract

This paper focuses on the computation of partial transitive closures (PTC) in deductive databases. We present an algorithm that takes advantage of topological sort and a tagging technique to achieve better performance. The algorithm has two phases. The first phase, like many others, is to generate a topologically sorted subgraph, containing only the reachable nodes and edges of the query. In the second phase, we evaluate the nodes from sources to leaves in topological order. Since the reachable subgraph is already sorted, the evaluation of nodes can be fully sequenced. A tag is associated with each node traversed, indicating from which source nodes the node can be reached. This tagging technique simplifies the storage of intermediate results and speeds up the computation. The new algorithm is simple, easy to implement, and yet efficient. It compares favorably to well-known algorithms, especially when the number of source nodes is not large.

Key Words : Deductive Databases, Transitive Closures.

1. Introduction

Much research on deductive database systems has been done in the past decade to meet the requirements of new database applications. The success of such systems largely depends on the efficiency of the recursive query processing techniques used. The complete transitive closure (CTC) of a directed graph is a binary relation, such that tuple (i, j) is in CTC if, and only if, there is a path from node i to node j. The partial transitive closure (PTC) of a given source set S is a subset of CTC that contains only those tuples (i, j), $i \in S$. It has been pointed out [11] that the computation of partial transitive closures is very common in recursive query processing. Jiang [6] has further suggested to implement PTC computation as an elementary database operation in new database systems.

In this paper, we present a new algorithm to compute PTC. The algorithm takes advantage of topological sort and a tagging technique to achieve better performance. The tagging technique simplifies the storage of intermediate results and speeds up the computation.

The rest of the paper is organized as follows. In Section 2, we first discuss some related work, and then introduce our algorithm in Section 3. In Section 4, we present the results of empirical performance evaluation of our algorithm. Section 5 is the conclusion.

2. Related Work

In this section, we define notations and discuss related work.

2.1 Notations

We seek to compute PTC of a directed graph G(V, E) for a given set of source nodes S, where V is the set of nodes and E is the set of edges in the graph G. We use VREACHG(i) to denote the set of nodes in G that are reachable from i, and EREACHG(i) to denote the set of edges in G that are reachable from i. VREACHG(i) is also called the reachability set of i. For simplicity, the subscript G in VREACHG(i) and EREACHG(i) is often omitted when there is no ambiguity. |VREACH(i)| and |EREACH(i)| denote the cardinalities of the respective sets, and |S| denote the cardinality of the source set S. SUC(i) is the set of immediate successors of node i.

The magic set of S [2, 5, 3], denoted M, is the set of nodes that are reachable from nodes in S. The magic graph GM for S [2, 3] is the part of graph G whose nodes are in the magic set M. In other words, the magic set and magic graph are the reachability set and

reachable graph of S, respectively.

2.2 Jiang's Algorithm

Jiang's algorithm [7] uses a combination of direct search and dynamic programming, in which breadth-first search is the dominant traversal strategy. The directed graph is stored as (immediate) successor lists (i.e., (parent, child1, child2, ...) lists) on disk. To avoid repeated traversals of common subgraphs, intermediate results (i.e., the reachability sets) of multi-input nodes (i.e., nodes with multiple incoming edges) are saved for potential reuse, as they may be visited again from other source nodes. The reachability set of a node can be obtained by taking a union of its reachable single-input nodes and the reachability sets of its descendant multi-input nodes.

It can be observed that the more multi-input nodes a graph has, the more the algorithm performs like dynamic programming. When all the nodes have multiple incoming edges, the algorithm becomes the dynamic programming. On the other hand, the algorithm degenerates to the direct search algorithm when all the nodes are single input nodes. The algorithm is an improvement over the dynamic programming because it does not save VREACH for any single-input node, which will not be visited from other source nodes. However, it also has a drawback. That is, some of the intermediate results saved with multi-input nodes may be of no use later because there is simply no path from other source nodes to them.

2.3 Jakobsson's Algorithm

It is assumed [5] that the magic graph has already been derived beforehand and stored as predecessor lists. The intermediate result is stored as a special-node tree for each node in the magic graph. A special-node tree is a predecessor tree containing only special-nodes, except probably for the root of the tree. A special-node is a source node or the nearest common successor of at least two special nodes. For each node in the magic set, the special-node tree is constructed by merging the special-node trees of its immediate ancestors.

Jakobsson's has the following drawbacks. The algorithm requires that the magic graph be derived first, while Jiang's doesn't. The storage of a special-node tree for each node in the magic graph may turn out to need more space than the reachability sets of Jiang's multi-input nodes. Moreover, the tree operations (i.e., searching and merging) of Jakobsson's may be more time consuming than the set operations of Jiang's.

2.3.3 BTC Algorithms

The algorithm computes PTC in two phases. First, the magic graph is extracted and sorted topologically. Then, in the second phase, PTC is computed on the derived magic graph. The nodes in the magic graph are processed in reverse topological order. The reachability set of a node is obtained by merging the reachability sets of its immediate successors, however, in topological order to avoid unnecessary set unions. For example, to compute VREACH(d) in Figure 2.1, a union with VREACH(f) should be performed before the union with VREACH(i) because node f precedes i in topological order. Since node i is also a successor of f, VREACH(f) must have already contained all the nodes of VREACH(i). Therefore, the union between

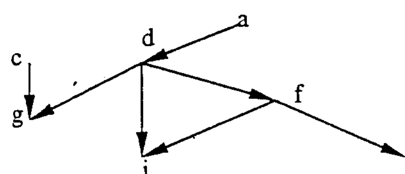


Figure 2.1. Marking Optimization

VREACH(d) and VREACH(i) can be omitted once we find out that successor i has already been added to VREACH(d). A redundant edge like e(d, i) can either be a cross or a forward edge [1]. This is referred to as marking optimization in [4].

As a simple comparison, BTC attempts to avoid unnecessary unions through the use of marking optimization, while Jiang's and Jakobsson's algorithms incorporate special mechanism (i.e., multi-input nodes, special-node trees) to achieve the same goal and reduce storage space for intermediate results. Both BTC and Jakobsson's require the extraction of the magic graph before traversing, while Jiang's doesn't.

3. The New Algorithm

Except for the base graphs, which are originally stored on disks, all other data structures, including magic graphs, intermediate results, etc., are stored in memory. This implementation assumption also applies to all other algorithms. The algorithm has two phases. The first phase is the same as that of BTC, that is, to generate a topologically sorted magic graph. In the second phase, we evaluate nodes from sources to leaves in topological order, instead of from leaves to sources in reverse topological order as in BTC. A tag is associated with each node traversed, indicating from which source nodes the node can be reached. The tag values directly translate into output tuples.

3.1. The First Phase - Preprocessing

The preprocessing phase generates an acyclic yet topologically sorted magic graph by applying Tarjan's algorithm [9]. Nodes in a cycle are collapsed into a single node [10] for efficiency. This preprocessing phase is the same as that of BTC. Although Jakobsson's and Jiang's do not specify such a preprocessing phase, they all use Tarjan's algorithm for more or less the same purpose. Specifically, Jakobsson's algorithm requires that cycles be removed beforehand since it assumed acyclic magic graphs. Jiang's algorithm deals with cycles using Tarjan's algorithm at run-time during PTC computation.

3.1.1 Storage Structure

The resulting magic graph is stored in the form of (immediate) successor lists [4, 7] and is here called the main table. A record in the table has three fields: Orderkey, (node-)ID, and SUC-list. The Orderkey, obtained in the topological sorting, stores the topological order key of the node; the ID specifies the name of the node; and the SUC-list stores the order keys of its immediate successors. The records are stored in topological order of nodes (i.e., Orderkey). Table 1 illustrates the table structure for the graph (assumed a magic graph) shown in Figure 2.1.

3.2. The Second Phase - Computation of PT

We attempt to explore reachable node, from sources to leaves, in topological order. Initially, only the source nodes are considered as (known) reachable nodes. We process a known reachable node, say, i that has the smallest order key (i.e., in topological order) by first retrieving its corresponding record from the main table. The immediate successors of the node thus become (known) reachable nodes. Meanwhile, we write output tuples (s, i) for all s ($s \in S, s \neq i$) that reach i. Such s can be obtained using a simple tagging technique (discussed shortly). We repeat the above process for each unprocessed reachable node in topological order until there is no unexplored reachable node. It can be observed that records are retrieved from the table in the same order as they are stored and thus yields minimal I/O if the table is stored on disk. In the following, we first introduce a tagging technique that is used for storing intermediate results as well as for computing PTC. And then we describe a simple mechanism that uses a queue to help sequence the retrieval.

3.2.1. A Tagging Technique

An array of bits, which we call a tag, is associated with each node, indicating from which source nodes the node can be reached. The algorithm uses tags to store intermediate results and produce PTC during traversal. Assume that there are k source nodes, say, s_1, s_2,..., s_k. Then, a tag with k bits will suffice to serve the purpose. Note that the intermediate result stores source nodes that reach the node in question, instead of reachable nodes of the node as in other approaches, e.g., BTC, Jiang's. This presents a unique advantage (over storing reachable nodes) when the number of source nodes is not large.

The rules for assigning values to tags are stated as follows. The ith bit, counted from left to right, of a tag associated with a node is set to 1, if and only if:

1. the node is the source node si (i.e., source node number i); or
2. the node is reachable from the source node s_i.

The first rule initializes the tag values of source nodes, indicating that a source node is reachable from itself. The second rule defines the tag values of ordinary reachable nodes.

Let's again consider the magic graph in Figure 2.1. Assume that c and a are source nodes, which are arbitrarily numbered as source node number 1 and 2, respectively. Note that the source node numbers 1, 2 assigned to c, a have nothing to do with their Orderkeys.

1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Since there are two source nodes, two bits are sufficient for each tag. For source nodes c and a, as they are the number 1 and 2 source nodes, their tags are initialized to 10 (i.e., the first bit is set) and 01 (i.e., the second bit is set), respectively, following the tagging rule 1.

Since node d is reachable only from source node a, its tag will eventually be set (discussed later) to 01 to that effect following rule 2. The tag value 01 associated with d, indicating that source node 2 has a path to d, can easily be translated into output tuples (a, d). By the same token, nodes f, i, and e will all eventually have their tags set to 01, which translate to output tuples (a, f), (a, i) and (a, e), respectively. Node g will have its tag set to 11 because it is reachable from both the source nodes c and a, which in turn translates into output tuples (s, g), where s = c, a.

Now, let us discuss how these tag values are computed during the traversal. When the record of source node a is retrieved from the main table, its immediate successor d is identified as a reachable node through the SUC-list. The tag of d is set to 01 by performing a bitwise OR operation on the tag of a (01) and the tag of d (00, initially), meaning that any source node that has a path to a also has a path to d. Similarly, when node d is processed, the tags of its immediate successors f, i, and g are all set to 01 by an OR operation on the tag of d (01) and the respective tags (all initialized to 00), indicating that they are reachable from a. Later on, when c is processed and g is found to be a successor, we modify g's tag to 11 by again performing an OR on c's tag (10) and g's tag (01). It can be conceived that the tag value of a node, obtained by bitwise OR operations on the tags of its parents and its own, reflects the fact that if a source node can reach the parent of a node, it can also reach the node, too. The tags store intermediate results and the tagging process is actually the process of computing PTC.

3.2. 2 Sequence the Search Using a Queue

It can be observed from the above discussion that the tag value of a node is computed from the tags of its immediate ancestors. Therefore, the tag value of a node is not completely set until all its immediate ancestors have their tags set properly. In an attempt to compute the PTC in a single traversal of the graph, the nodes must be visited in their topological order.

For simplicity, we assume that Orderkeys take consecutive integers from 1 to |M| (the size of magic set). The queue can be easily implemented as an array of pointers pointing to the tags, with the ith element of the array designated to a node whose Orderkey is i.

```
tag = array [1.. k] of bit;        /* k = |S| */
Q: array [1.. |M|] of (pointer to tag);
```

We assume that initially all elements of Q point to clean tags (all bits set to 0). The tag values of source nodes are to be set first following rule 1. Nodes (or elements) in the queue are processed one by one in order. As a node, say, i becomes the head of the queue, we retrieve the record for i from the main table. The immediate successors of i are identified, and their tag values are obtained by an OR operation on the node i's tag and their respective ones, reflecting the fact that source nodes that can reach i can also reach its successors. Note that no records for the successors are actually retrieved from the main table until the corresponding nodes become the head of the queue. This ensures topological traversal of the magic graph.

Whenever a node i becomes the head of the queue, all of its ancestors must have been processed completely, and its own tag has also been set completely. The tag of i indicates from which source nodes node i can be reached. We can translate the tag value into output tuples of the form (s, i), where s is a source node whose corresponding bit in the tag is set. After the queue is processed completely, the set of output tuples is exactly the PTC we are trying to compute. Note that no duplicates will ever be produced, no records will be retrieved twice from the main table, and no disk blocks will ever be read twice if the main table is stored on disk.

3.2.3 The Algorithm

The algorithm is presented in Procedure FindPTC. For ease of presentation, we assume that procedures Newtag() and GetRecord(y) are already implemented. Newtag() allocates space for a tag and initialize it. GetRecord(y) is to read the record for node y from the main table.

```
Procedure FindPTC (S: source set)
{       tag = array [1..k] of bit;
        Q: array [1..|M|] of (pointer of tag);
        i, y, orderkey: integer;
        s, si : ID;
```

1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

```
    For (i := 1 to |M| )   do  Q[i] := Newtag();
    For (each node  si  in S)    do
         set  the  ith  bit  of   si 's  tag to 1;
    For (orderkey := 1  to  |M|)    do
    {
         r := GetRecord (orderkey);
         For each bit set to 1 in *Q[orderkey]     do
           /* let s be the corresponding source node*/
              if (s   r.ID)  write an output tuple (s, r.ID);
         For each Orderkey y in r.SUC-list          do
            /* y is an orderkey of a node */
              *Q[y] := *Q[y]  |  *Q[orderkey];
      /* a bitwise OR on its parent and its own tags */
         }
}
```

Theorem 1: The algorithm computes PTC correctly for a given set S of source nodes.

Proof sketch : The tag bit of a node r corresponding to a source node s in S is set if, and only if, there is a path from s to r.

Theorem 2: The algorithm has a complexity of $\Theta(|S|.|E_M|)$, where $|E_M|$ is the number of edges in the magic graph.

Proof sketch : An OR operation is performed on tags (of size $|S|$ each) for each edge in the magic graph.

3.3. Comparisons with BTC Algorithm

The BTC algorithm deserves special attention because it bears the most similarity to our algorithm and has generally better performance than other previously mentioned algorithms [3]. While both BTC and ours have exactly the same first phase, i.e., generating topologically sorted magic graphs, significant differences appear in the second phase. Our algorithm explores nodes from sources to leaves in topological order, while BTC explores from leaves to sources in reverse topological order. In BTC, the set of reachable nodes of a node is obtained by performing union operations on sets of reachable nodes of its children. Instead, ours computes the set of source nodes that reaches a node by performing union operations (bitwise ORs) on the sets of source nodes that reach its parents. The union operations on reachable node sets can be extremely expensive, especially when the sets are large. According to our careful study, it shows that it would be most computationally efficient to represent sets of reachable nodes as bit vectors, like tags of ours, and use bit ORs for the unions. Therefore, reachable nodes (or intermediate results) in both BTC and Jiang's will all be implemented as tags for comparisons. This uniform implementation adds another resemblance between BTC and ours, and makes comparisons simple. However, the size of our tags is $|S|$, while it is $|M|$ for both BTC and Jiang's. Note that $|S| \le |M|$ and this could have significant impact on the performance. On the other hand, BTC may perform less ORs by identifying cross and forward edges. Essentially, BTC has a complexity of $\Theta(|M|.|E_{M'}|)$, where $|E_{M'}|$ is the number of edges, excluding cross and forward edges, in the magic graph. It can be observed that $|S|$, $|M|$, and $|E_{M'}|$ are the major factors determining the performance.

4. Preliminary Experimental Results

The experiments are run on Sun Sparc5, with 110 MHZ speed and 32MB memory.

4.1. Assumptions

For simplicity, we shall assume, as in [3], that all algorithms have the same preprocessing phase (like BTC and ours) of generating a topologically sorted magic graph in the form of successor or predecessor lists. Although Jakobsson's does not require a sorted magic graph, it assumes an acyclic magic graph, which requires essentially the same amount of work as obtaining a sorted magic graph, yet it can greatly benefit from using a sorted magic graph by sequencing the search in the graph. As for Jiang's, it simply moves the job of removing cycles from run-time to the preprocessing phase. Moreover, studies [3, 8] have shown that the cost of a topological sort is negligible and thus is often ignored. As a result, we believe such an assumption of a uniform preprocessing phase should not yield any unfairness in the comparisons.

4.2 Query Parameters

We generate synthetic graphs in the same way as [3] to study the performance of the algorithms. An acyclic, topologically sorted graph is generated by first numbering the nodes and then adding arcs from low

numbered nodes to high numbered nodes. A graph is characterized by its number of nodes, average out-degree (average number of outgoing edges of nodes), and locality, which is defined to be the maximum difference between pairs of node numbers of edges.

4.3 Experimental Results

We consider CPU time and I/O of all algorithms. I/O cost for reading magic graphs is not discussed due to our simplifying assumption that the magic graph is already stored in memory. In addition, the cost of writing output tuples will also not be included in the comparisons as each algorithm writes the same amount of output tuples. Due to space limitation, interested readers are referred to [8] for more comprehensive experimental results.

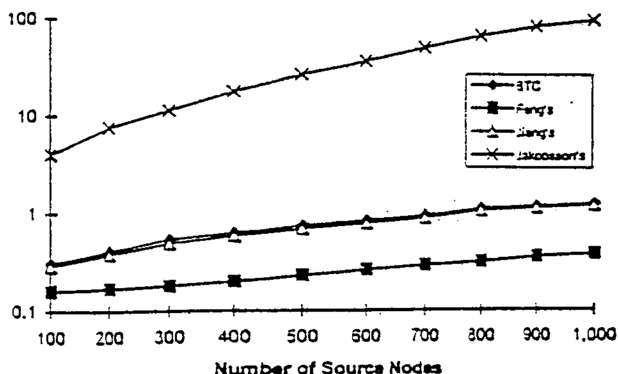
4.3.1 CPU Time

The CPU cost not only has to do with the number of operations performed but also the complexity of the operations themselves. Consequently, the storage of intermediate results and the operations on them can have great impact on the performance. As mentioned earlier, we have chosen bit vectors (tags) to store intermediate results of Jiang's, BTC, and ours as it is most efficient for the dominant union operations. Jakobsson's has its own data structure for special-node trees, which could not be replaced by tags effectively. Therefore, we will stick to Jakobsson's original data structures.

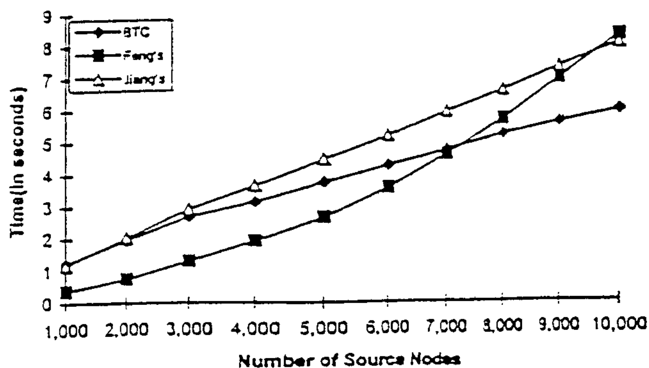
In general, our algorithm is the simplest, BTC is the next, and Jakobsson's is most complex. Recall that while a tag in BTC and Jiang's requires |M| bits, ours needs only |S| bits ($|S| \leq |M|$). However, BTC and Jiang's algorithms may perform less ORs than ours due to their respective optimization measures. Specifically, Jiang's algorithm performs set unions only on intermediate results of multi-input nodes, while BTC does not perform unions on forward and cross edges.

In Figures 4.1.(a) and (b), we show the CPU cost of the algorithms on average out-degree value 2. The graphs have 10,000 nodes with a locality 10,000. It is observed that Jakobsson's algorithm performed much slower in all cases than others. This is mainly due to its complex representation of special-node trees (versus the bit vectors), expensive tree merging operations (versus the OR operations), and extensive use of stacks. The special-node trees become large as the number of source nodes and average out-degree increase, and thus increase the cost of merging rapidly. Moreover, since Jakobsson's derives a superset of the PTC, extra computation is incurred in deriving such extra tuples and later removing them.

As shown in Figure 4.1.(a), our algorithm clearly outperformed Jiang's and BTC when the number of source nodes is not very large (or when |S|/|M| is small, to be more precise). This is mainly due to the smaller size of our tags (|S| << |M|). Jiang's algorithm did perform slightly better than BTC when the out-degree is low. This is mainly due to the fact that the graphs have a relatively larger number of single-input nodes than forward/cross edges when the out-degree is low. However, much of the gain is offset by its complex stack management.



(a) Small Numbers of Source Nodes



(b) Large Number of Source Nodes
Figure 4.1 CPU Time

As illustrated in Figures 4.1, when the number of source nodes increases, |S|/ |M| ratio is getting closer to 1, the advantage of smaller tags dwindles, and the performance of BTC catches up. As shown in Figure 4.1.(b), BTC outperformed ours when the numbers of source nodes are greater than 7,000 (out of 10,000). This is mainly due to the fact that there are around 15% of the edges in the graphs (of average out-degree 2) are forward or cross edges, and this outweighs the gains of smaller tags and less overhead (due to the simplicity) of our algorithm.

4.3.2 Page I/O

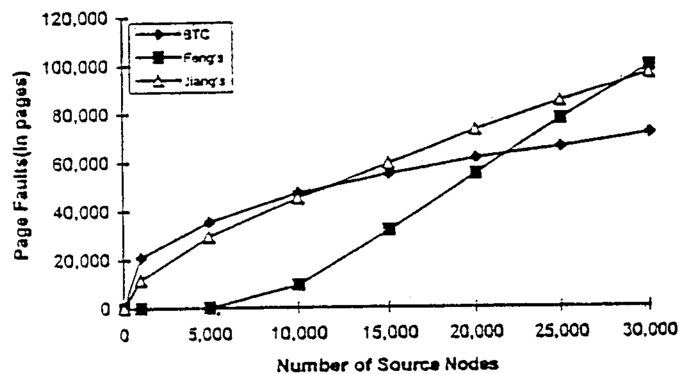
Space requirement also has a great impact on the performance of an algorithm. If the data structures of an algorithm can not be accommodated in memory, page faults could result and thus degrade the performance. The more space an algorithm needs, the more likely page faults will occur.

BTC and our algorithm use the same and the simplest data structures, bit-vectors and arrays, and have the same access pattern. Any page replacement policy is expected to have the same effect on both algorithms. Jakobsson's has the most complex tree structures and operations, and uses stacks extensively. It might be quite tedious for a simulated buffer manager to keep track of all the activities. Moreover, since page and list replacement policies have only secondary effects on the performance [3], we will not implement a simulated buffer manager and simply let the OS manage the available space. The underlying page replacement policy of the system is LRU. We generate graphs with 40,000 nodes, which are large enough to generate large numbers of page faults to view the trend.

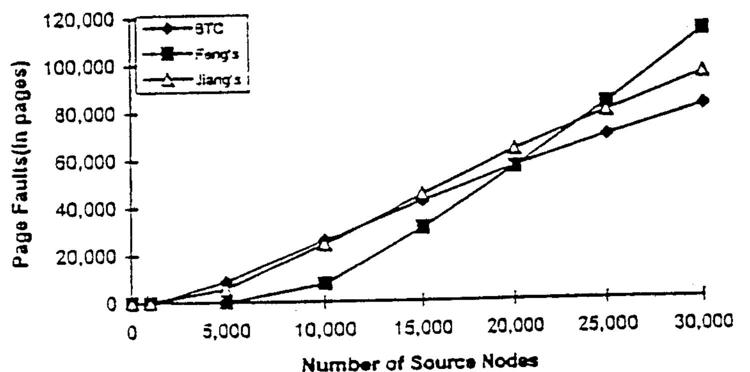
The main data structure in our algorithm is an array of pointers and tags. The space required for the array is |M| . sizeof(pointer) and the space for tags is |M|. |S| bits.

BTC essentially uses the same data structures as ours, however, with tags of size |M| bits. In Jiang's algorithm, the intermediate result (tag) for each source node and each node with multiple incoming edges, is to be kept. That is, there are $|S \cup V_{multi\text{-}in}|$ reachability sets, where S is the set of source nodes and $V_{multi\text{-}in}$ is the set of nodes with multiple incoming edges. The size of each tag is |M| bits. It is worth mentioning that if we had not assumed a preprocessing phase for Jiang's algorithm, it could have needed tags of size N bits, where N is the number of nodes in the entire graph. In addition, each source node or multi-input node needs to have a problem list storing descendant multi-input nodes. The algorithm also needs a stack, an array of |M| integers, and two |M|-bit vectors indicating node types (leaf nodes, multi-input nodes).

Jakobsson's algorithm needs to store a special-node tree with each node traversed. It also employs a bit-vector to record the set of nodes for which the special-node trees have been computed. Note that the structure of the special-node tree is more complex than the tags.



(a) Locality : 5,000 Average Out-degree : 2



(b) Locality : 40,000 Average Out-degree : 2

Figure 4.2 Page I/O

It is obvious that our algorithm uses less space than BTC due to smaller tags. The space requirement for

Jiang's is dependent upon the particularity of the graph. If few nodes have multiple incoming edges, it may have good performance. As for Jakobsson's, if there are many special-nodes (quite often that is the case when the number of edges is large), the algorithm may require a very large amount of memory space.

Figure 4.2 shows the page I/O of each algorithm. For simplicity, we show only the case where the average out-degree is 2. Locality also has some effect on the performance. Jakobsson's requires lots of space for special node trees and thus generated extremely large numbers of page faults. Its results will not be discussed further.

Our algorithm initially performs best. While other algorithms have large numbers of page faults, ours has only few or no page faults initially. This is due to the fact that our algorithm requires less space than others. As the number of source nodes increases, more space is required, and more and more tags could not be stored in memory. Since BTC performs less unions of which some can cause page faults, page fault rates of our algorithm increases faster than BTC. BTC eventually has a better performance when the size required for intermediate results is much larger than the memory space. As the number of source nodes increases beyond 30,000, the system essentially thrashes for all algorithms due to the lack of memory.

Locality also has some effects on the performance. An edge that links two nodes far apart (in terms of their topological order keys) is more likely to generate a page fault. As shown in Figure 4.2.(b), higher page fault rates are observed when the locality is larger (locality=40,0000) than in (a) & (c) (locality=5,000), respectively.

5. Conclusion and Future Work

In this paper, we have proposed a new algorithm for computing PTC. The algorithm is so designed that nodes are visited in topological order. The new algorithm is conceptually simpler and easier to implement than others. The tagging technique allows us to store and derive intermediate results efficiently. In addition, the algorithm also requires less space than others. In general, it has very good performance when the number of source nodes (compared to total number of nodes) is not very large. It is most suitable for small to median-sized PTC queries.

References

[1]] A. Aho, J. Hopcroft and J. Ullman, Data structures and algorithms. Addison-Wesley, 1983.

[2] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman, Magic sets and other strange ways to implement logic programs, Proc. ACM PODS, 1-15, 1986.

[3] S. Dar and R. Ramakrishnan, A performance study of transitive closure algorithms. Proc. ACM SIGMOD Conf. Management of Data, 454-465, 1994.

[4] Y. Ioannidis, R. Ramakrishnan, and L. Winger, "Transitive Closure Algorithms Based on Depth-First Search", ACM TODS, to appear.

[5] H. Jakobsson, On tree-based techniques for query evaluation. Proc. 11th ACM Symp. Principles of Database Systems, 380-392, May 1992.

[6] B. Jiang, Making the partial transitive closure an elementary database operation. Proc. GI Conf. Database Systems for Office Automation, Engineering and Scientific Applications, 1989.

[7] B. Jiang, A suitable algorithm for computing partial transitive closures in databases. Proc. IEEE Conf. Data Engineering, 264-271, 1990.

[8] M. Qian, Performance evaluation of partial transitive closure algorithms, M.S Thesis, SIU, January, 1997.

[9] R. Tarjan, Depth-first search and linear graph algorithms. SIAM J. Comp., 146-160, June 1972.

[10] M. Yannakakis, "Graph-Theoretic Mehtods in Database Theory", Proc. 9th PODS, Nashville, Tennessee, 230-242, April 1990.

[11] C. Youn, L. J. Henschen, and J. Han, Classification of recursive formulas in deductive databases. Proc. ACM SIGMOD Conf. Management of Data, 320-328, June 1988.