New Transitive Closure Algorithm for Recursive Query Processing in Deductive Databases

I. H. Toroslu
Dept. of EECS
Northwestern University
Evanston, IL 60208
E-mail: toroslu@eecs.nwu.edu

G. Z. Qadah
AT&T Bell Labs
480 Red Hill Road
Middletown, New Jersey 07748
E-mail:qad@uhura@at.com

Abstract

The development of efficient algorithms to process the different forms of the transitive-closure (TC) queries within the context of large database systems has recently attracted a large amount of research efforts. In this paper, we present a new algorithm suitable for full transitive closure problem, which is used to solve uninstated recursive queries in deductive databases. In this new algorithm there are two phases. In the first phase a general graph is condensed into an acyclic graph and at the same time a special sparse matrix is formed from the acyclic graph. The second phase is the main one where all of the page I/O operations are minimized. Using simulation, this paper also studies and examines the performance of this algorithm and compares it with the previous algorithms.

1 Introduction

The development of efficient algorithms to process the transitive closure (TC) rules and queries within the context of large database systems has recently attracted a large amount of research efforts [1, 6, 7, 8, 10, 11]. This is due to the important role of the TC rules in improving the intelligence of database systems. In general, a logic rule is linearly recursive if the rule’s head predicate appears only once in its body. A TC rule is a linearly recursive rule of the following form:

\[ R(X, Y) : -A(X, Z), R(Z, Y) \]  \hspace{1cm} (1)

where \( A(X, Z) \) is an extensional (base) predicate. Within the context of deductive databases, \( A(X, Z) \) is defined by a two-attribute normalized database relation with very many tuples as shown in Figure 1a.

Another common view for the base relation is the one shown in Figure 1b, where the base relation is represented as a directed (tree, acyclic or cyclic) graph (with the possibility of having more than one component). The nodes in such a graph are the set of distinct values in the two columns of the base relation. For every tuple \( < x, y > \) of the base relation, there exists, in the corresponding digraph, a directed edge from node \( x \) to node \( y \).

To generate solutions from the above recursive rule, another non-recursive rule, the exit rule, which defines the predicate \( R(X, Y) \) must exist. For example, the following rule is an exit rule for relation \( R \):

\[ R(X, Y) : -A(X, Y) \]  \hspace{1cm} (2)

A TC query is a headless rule whose predicate is defined by a TC rule and its associated exit rule. For example,

\[ : -R(X, Y) \]  \hspace{1cm} (3)

is a TC query. In general, a 2-place unit query, such as (3), may have different forms depending on the instantiation status of the query’s variables [8]. In this paper, we propose an algorithm for solving the full TC problem which corresponds to the TC query form where both of the query variables are uninstatedented.

A number of algorithms have been developed to compute the full transitive closure (FTC) problem within the database context. In that context, the graphs representing relations are assumed to be too large to fit into main memory, and therefore the most important consideration of the developed algorithms is the reduction of the page I/O traffic between main memory and the secondary memory. The FTC problem is a well known and a classical problem in graph theory. Therefore, several main memory algorithms (i.e., the graph fits to main memory) have been devel-
Table 1: The Relation A

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>e</td>
</tr>
<tr>
<td>g</td>
<td>d</td>
</tr>
<tr>
<td>f</td>
<td>d</td>
</tr>
<tr>
<td>f</td>
<td>b</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

Figure 1: The Binary Relation A.

a) In Table Form.

b) In Graph Form.

Opined to compute the transitive closure of the graph efficiently. Some of those algorithms have been adopted to a database environment and a lot of new algorithms have been developed special for the database transitive closure problem. Basically, these algorithms can be categorized into two groups: iterative and direct [1].

**Iterative algorithms**, as their name implies, solve the transitive closure problem by computing a relational algebraic expression(s) repeatedly. Because the relation size is finite, after a finite number of iterations have been performed, no more new tuples will be generated. Therefore, for these algorithms the processing time (and the number of iterations also) depends on the characteristics of the underlying database relation. Examples of these algorithms are the semi-naive and the logarithmic algorithms [2, 11].

In **direct algorithms**, however, each element (a node or an edge) is processed a constant number of times. Therefore, the termination of the algorithms are independent of the underlying database. The algorithms stop when the processing of all the elements in the graph are completed. Two types of direct algorithms exist, namely matrix-based and graph-based.


**Graph-based direct algorithms**, on the other hand, are defined in terms of graph traversal operations on the relation graph. Usually, graph-based algorithms are developed for acyclic graphs and therefore, cyclic graphs are converted into acyclic ones by condensing the strongly connected components into one node. Some graph-based direct algorithms are [3, 4, 6].

Although a large number of algorithms have been developed to solve the FTC problem, we will present and discuss only one of them, the hybrid algorithm of [1], since it has been shown in [1] that this algorithm achieves the best performance among all of its competitors. Therefore, in section 2 we will present the hybrid algorithm and discussion of its drawbacks. Section 3 presents the new algorithm. In section 4, we compare the performance of these two algorithms and show the superiority of the new algorithm over the hybrid one. Finally, section 5 presents some concluding remarks.

## 2 The Hybrid Algorithm

One of the most recent algorithms developed by Agrawal and Jagadish [1], is called the **hybrid algorithm** because it combines the matrix representation
of the relation together with the graph traversal operations on the relation's directed graph to solve the FTC problem.

In the hybrid algorithm, an adjacency matrix is created for the processed graph using a topological ordering on the nodes instead of the arbitrary ordering of the nodes in the matrix. To be able to do the topological ordering on a cyclic graph, the graph must first be converted into an acyclic one. In the first pass of the hybrid algorithm, the strongly connected components are identified and coalesced into single nodes (using Tarjan's algorithm of \([9]\)) and at the same time a topological sorting of the condensed graph is performed. In the second pass, the hybrid algorithm is used to compute the transitive closure on the topologically sorted acyclic graph.

To explain how this algorithm works, we will skip the first pass and present the second pass on an acyclic graph. For the graph of Figure 1, the first matrix in Figure 2 (this is a binary matrix representation of the graph, but indeed each node is stored with the list of its immediate successors where these successor lists are using the topological order shown in the matrix) will be created by the first pass. The second matrix in this figure shows the transitive closure of the graph. The algorithm for the second pass is presented in Figure 3.

For example, the successors of the node \(f\) are computed as follows (refer to Figure 2): since all the rows with index less than \(i\) are processed before row \(i\), it is guaranteed that all \(i\)'s immediate successors have been processed completely (immediate successors of the node \(i\) correspond to some rows \(j\) where \(j < i\)) before its process. Therefore, before processing node \(f\), the successors of its immediate successors \(d, b\), and \(a\) have already been computed. These immediate successors will be searched from higher order to lower (i.e., the row is processed from right to left).

First, node \(d\) will be processed. It has two successors, \(a\) and \(c\). They must be added to the successor list of \(f\). However, because \(a\) is already in the successor list of \(f\) (it is an immediate successor of \(f\)), only \(c\) is added. Also, to optimize the process (marking optimization), node \(a\) is removed from the temporary immediate successor list of \(f\) to prevent it from being reprocessed. After that, the second highest immediate successor, \(b\), is processed. It has only one successor, \(a\), and that will be added to the successor list of \(f\). This time \(a\) is added (for the second time) to the successor list of \(f\) as an unwanted result of marking optimization operation (i.e., it was removed from the temporary immediate successor list). The last immediate successor of \(f\) is \(a\). Because \(a\) was removed from the temporary immediate successor list of \(f\) as the result of the marking optimization operation, it will not be reprocessed.

Totally six nodes are processed, where \(f\) has only four successors. Therefore, there are two redundant operations. Figure 2 shows the number of nodes processed for each successor lists. Numbers in parenthesis show the number of nodes processed for each immediate successor. The final transitive closure has 17 tuples, but, hybrid algorithm processes 23 nodes from

---

**Figure 2:** Hybrid Algorithm Matrix Representation.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>g</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) + 1</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) + 1</td>
<td></td>
<td></td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0 + 1) + 2</td>
<td></td>
<td>d</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1 + 2) + 2</td>
<td></td>
<td>e</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0 + 0 + 4) + 3</td>
<td></td>
<td>g</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(0 + 1 + 2) + 3</td>
<td></td>
<td>f</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3:** The Hybrid Algorithm.

```plaintext
procedure hybrid_algorithm()
    for i = 1 to n
        copy row i into a temporary i*
        for j = i - 1 downto 1
            // process from right to left in row */
            if (i*, j) ≠ 0
                /* immediate successor opt. */
                add_succ(i, j, i*)
                /* add successors of j to i */
        end
    end

procedure add_succ(i, j, i*)
    for k=l to j-1
        if (j, k) = 1
            if (i*, k) = 1
                (i*, k) = 0 /* marking opt. */
            else
                (i, k) = 1
            end
        end
    end
```


the matrix by making 6 redundant operations.

In [1], blocked version of the algorithm is also proposed. Agrawal and Jagadish have also compared the performance of the blocked version of the hybrid algorithm with the grid algorithm of [10] and the graph-based algorithm of [6] and showed that the hybrid algorithm has the best I/O performance.

3 The New FTC Algorithm

Although, the hybrid algorithm has been shown to outperform any of its competitors in computing the FTC problem, it still involves some redundant computations such as trying to add already existing tuples into the transitive closure and some redundant accesses to the matrix nodes which do not create new tuples for the transitive closure. An algorithm which performs better than the hybrid algorithm can be developed by avoiding all of these redundant computations and accesses to the matrix and the transitive closure.

Similar to the hybrid algorithm, the algorithm to be presented here includes an initial phase where the general graph is condensed into an acyclic graph. Such a graph reduction is achieved by using a depth first search procedure which stores the result acyclic graph in a special matrix form.

The structure used in our algorithm is a special matrix in which rows represent the paths in the graph. Basically, depth-first search is used to create the paths of the graph. In a simple graph with no backward, cross or forward edges (a simple tree), paths can be stored as a sequence of nodes starting from the root node (the node with no incoming edges) and ending at the leaf nodes. However, if the paths are directly stored some of the nodes must be repeated for different paths. Instead of storing every node of all of the paths, the common parts of these paths can be stored only once to avoid duplications. If two paths \( P_1 = < a_1, ..., a_n, b_1, ..., b_m > \) and \( P_2 = < a_1, ..., a_n, c_1, ..., c_l > \) have the common parts \( a_1, ..., a_n > \), then \( P_1 \) and \( P_2 \) can be stored in the two consecutive rows of the matrix in which the first \( n \) entries \( a_1, ..., a_n \) are the same and \( a_n, b_1, ..., b_m > \) and \( a_n, c_1, ..., c_l > \) where the first \( n \) entries of the second row is empty. Simply each row represents a path starting from the first column of the row. The empty entries at the beginning of the rows inherit the paths from the non-empty entries of their nearest upper row(s).

In this way, trees (the simplest forms of graphs) can be stored in the matrix form without repeating
any node. The matrix in Figure 4b shows the matrix representation of the tree of Figure 4a created by doing a depth-first search on the tree starting at the root node a. The last row of this matrix represents the path < a, e, g > (e comes from the 3rd row and a comes from the 1st one).

For complicated (acyclic) graphs with forward, and cross edges some more information must be added to this structure in order to prevent the repetition of nodes in the matrix structure. Depth-first search visits each edge once and one entry is created for each edge in the matrix for the corresponding depth-first tree (plus one entry for each root node). When an edge is visited, the destination node is entered into the matrix. This is simple for the tree case, because, each node has only one incoming edge. Therefore, each node is visited only once and entered into the appropriate location in the matrix. However, if there are more than one incoming edge to a node, the depth-first search will visit the same node more than once (once for each one of the incoming edges). In such a case, to prevent the duplicate storage of the nodes in the matrix, a different technique is used; for the first visit to the node, it is entered into the matrix, but, for other visits, instead of the node itself, its coordinates in the matrix is entered into the matrix (like a pointer to the already stored node). In this way we guarantee that only a single copy of each one of the graph's nodes is entered into the matrix. Moreover, we guarantee that only one entry (either a node or a pointer) is created in the matrix for each edge in the graph.

In Figure 4, a complex graph (the graph in Figure 4c) and its associated matrix (Figure 4d) is presented. The graph has forward (a,d) and cross (f,d) edges. These edges are represented by pointers in the matrix of Figure 4d. In this graph there are 7 edges and in its matrix representation there are 7 + 1 = 8 non-empty entries (one for root node a).

In the implementation of this sparse matrix, the empty entries do not need to be explicitly stored. The matrix can be stored sequentially row by row. For each row, storing the column number of its first non-empty entry and the sequence of non-empty entries in the row is sufficient. Thus, the size of the matrix implementation is much smaller than the original relation.

After the special matrix form is created, finding the transitive closure of the graph involve the processing of each node one by one to compute the successor lists. The matrix is searched sequentially from top to bottom. If the entry currently searched has a node, (pointers are skipped) say nodei, with coordinates (rowi, coli), then its successors are determined as follows: starting from the next non-empty entry the matrix is scanned until reaching a row which starts with a column whose number is smaller or equal to coli. During this scan, all of the nodes found are directly added to the successor list (succi) of the node nodei. All of the successor lists, which form the transitive closure of the processed graph, are stored on disk. Successor lists can be created sequentially as the nodes are added to them. Meanwhile, pointers pointing to the coordinates above (rowi, coli) are added, without repetitions, to the pointer list (pointi) of nodei in ascending order. Therefore, pointer lists will be sorted using the coordinates they are pointing. Because pointer lists are very small, as to be seen in the next section of the paper, these lists are kept in the main memory.

When the scanning of one segment of the matrix is over, the pointer list of nodei, pointi, is processed to find the rest of the successors of nodei. Because the processed graph is restricted to be acyclic, pointers in pointi may represent forward or cross edges (but not the backward ones). Forward edges are not processed since they do not generate any additional nodes to the successor lists (since they do not point to coordinates above the node's own coordinate). One pointer can point to a node which is a successor of a node pointed by another pointer. In this case, processing the second pointer will be sufficient, because it will scan the other one and its successors anyway. To do this, the pointer that will be processed next is selected as follows: all of the pointers in pointi are scanned from top to bottom to find the topmost one where all of the pointers below it are successors. It is easy to determine whether a coordinate is a successor of another one. A coordinate (rk, ck) is a successor of another one (rj, cj) if:

- either rj = rk and cj < ck;
- or \( \forall (r_j) \) where \( r_j < r_k \leq r_k : \) first non-empty entry in \( r_i > c_j \).

For this computation another small array is used which has the column numbers of the first non-empty entries for each row (called row beginnings). This array is scanned as shown above to find whether a pointer is a successor of another one.

After the pointer is selected, that pointer and all of its successor pointers are removed from the pointer list. Then, the processing of the node pointed to by the selected pointer is done in the same way as it is done for the original node nodei. The only difference is, the node itself is also added to successor list succi, together with its own successors. Meanwhile the new pointers
procedure \texttt{ftc-algorithm}() 
\begin{verbatim}
for (z, y) = (1, 1) while not EOF do {
switch (type of matriz[z, y])
    POINTER:
        y = y + 1;
    END-ROW:
        z = z + 1;
        y = row.begin(x);
    NODE:
        i = matriz[z, y].node;
        point\_i = \emptyset;
        n\_pointers = 0;
        find.successors(i, x, y);
        while point\_i \neq \emptyset do {
            pointer\_i = select.pointer();
            (zp, yp) = pointer\_i.coord;
            succ\_i = matriz[zp, yp].node
            find.successors(i, 2, y);
        }
        y = y + 1;
    }
end}
\end{verbatim}

procedure \texttt{find.successors}(i, x, y) 
\begin{verbatim}
for (z, y) = (i, yi + 1) while y > yi do {
switch (type of matriz[x, y])
    POINTER:
        if (matriz[x, y].pointer \notin point\_i) and
            (matriz[x, y].pointer.coord < (x, yi))
        then {
            point\_i = point\_i \cup sorted matriz[x, y].pointer;
            n\_pointers = n\_pointers + 1;
        }
        y = y + 1;
    END-ROW:
        z = z + 1;
        y = row.begin(x);
    NODE:
        succ\_i = succ\_i \cdot matriz[x, y].node;
        y = y + 1;
end}
\end{verbatim}

procedure \texttt{select.pointer}() 
\begin{verbatim}
for k = 1 while k \leq n\_pointers do {
    flag = TRUE;
    for j = point\_i.k, row + 1
        while j \leq point\_i[n\_pointers], row do {
            if row.begin(j) \leq point\_i.k.col then {
                flag = FALSE;
                k = k + 1;
                break;
            }
        }
        j = j + 1;
    }
    if flag = TRUE then {
        n\_pointers = k - 1;
        return(point\_i[k]);
    }
end}
\end{verbatim}

Figure 5: The New FTC Algorithm.

Figure 6: Matrix Representation.
are visited and no redundant additions are performed.

4 Comparison of the Two Algorithms

The main purpose of the database query processing algorithms is to minimize the page swaps between the main memory and the disk. Because the relation and the resulting transitive closure are too large to fit into main memory, they are stored on disk. During the computation, they must be brought into main memory in pages. To minimize the page swaps between main memory and the disk, all the accesses to the relation and to the transitive closure must be minimized.

To quantitatively compare the performance of the hybrid algorithm and the newly proposed algorithm we have created random acyclic relations and compared the second phases of these two algorithms (since the first phases are almost identical). In the simulation program, Relation Size (or the number of tuples in the relation, N) is fixed to 1000. Main memory size allocated to the transitive closure computations is assumed to be \(\frac{1}{10}\)th of the size of the whole relation.

For each out-degree, ranging from 1 to 5, 10 different random relation are created and both of these algorithms are applied to find the corresponding transitive closures. "Out-degree" of the relation graph is defined as a ratio of "number of tuples" to "size of the attribute domain". If we show the size of the attribute domain as D, and out-degree as O, then \(O = \frac{N}{D}\).

To get results for different O values, by using fixed N, D is computed first and random relation is created using elements of the domain of size D. In our simulation, we obtained results for out-degrees between 1 to 5. Out-degree smaller than 1 is very trivial and results are similar to out-degree 1. Also, out-degrees greater than 5 will have similar patterns as out-degrees close to 5.

Normally the performance of the database algorithms are determined by measuring the number of page I/O's between main memory and disk. However, in comparing the hybrid algorithm and our algorithm we do not need that kind of measurement. These two algorithms have some similar computations. Basically there are two operations creating these page I/O's in the second phases of the hybrid algorithm and our algorithm. They are accesses to the matrices and accesses and writes to the transitive closures.

As it can be seen from the explanations in previous sections our algorithm has no redundant access to the matrix (which corresponds to the relation in the second phase of the algorithm) something that the hybrid algorithm cannot guarantee. Our algorithm has only as many node accesses as the number of tuples in the transitive closure. However, the hybrid algorithm has some redundant accesses to the matrix which increases when the outdegree is increased. In the examples of previous sections, the transitive closure has 17 tuples and the hybrid algorithm visits 23 nodes in the matrix, but, our algorithm visits only 17 necessary nodes. Of course, in addition to them it also visits some pointers, but, because the size of pointers are very small and the pointer lists are kept in main memory they have very small effect on the performance of the algorithm.

Figure 7 shows the comparison of two algorithms as the number of accesses to the matrices. Because the number of nodes accessed in the matrices for the new algorithm are equal to the actual number of tuples in the FTCs of the relations, the differences between these two algorithms in this figure show the redundancies (redundant accesses to the matrix) of the hybrid algorithm.

Another disadvantage of the hybrid algorithm is its access to partially computed transitive closure to find the successor lists. In our algorithm, however, for the computation of the successor list we only need to use the special matrix, which is even smaller than the relation itself and stored and processed sequentially. However, the hybrid algorithm needs to use the already computed successor lists which are stored as the part of the transitive closure during the computation. Because the size of transitive closure gets larger, it is...
stored on disk and each access to it would generate a large number of page I/O between main memory and the disk.

Because the hybrid algorithm needs to access the transitive closure for the computation of the successor lists and it might add the same tuple into the transitive closure more than once, the transitive closure must be stored using some random access storage structure by creating a lot of page swap operations between the disk-based transitive closure and main memory for each tuple added. Most of the extra accesses to the matrix create redundant transitive closure tuples and therefore redundant write operations to transitive closure in the hybrid algorithm. However, in our algorithm each tuple is created only once, and therefore, these tuples can directly be added to the transitive closure in a sequential manner.

The maximum pointer list size in our algorithm is less than the 10% of the number of tuples in the processed relation for the outdegrees between 1 and 5. If we assume that the size of each pointer as 2 + 2 = 4 bytes and each tuple as 16 + 16 = 32 bytes, the size of the maximum pointer list will be less than 2% of the size of the relation. In addition to the pointer list, we also need to keep row beginnings (i.e., the column number of each row's first non-empty entry) in the main memory to find the next pointer to be processed. This array will also be very small (1 byte for each row beginning, and the number of rows in the worst case can be equal to the number of tuples, so, the size of this array will be smaller than 3% of the size of the relation even in the worst case). Therefore, we can easily keep these two structures in main memory.

5 Conclusion

The computation of transitive closures is one of the most frequently needed operations in query processing in deductive databases. Therefore, a large volume of research has been devoted to solve this problem and as a result, a good number of new algorithms have been developed. The latest algorithm proposed in [1] has been shown to possess better performance over its competitors. However, still it has some drawbacks. In this paper, we have proposed a new transitive closure algorithm for deductive database query processing. Our algorithm overcomes all of the drawbacks encountered by the hybrid algorithm. We have also compared the new algorithm with the hybrid algorithm and showed the superior performance of the new algorithm relating in that of the hybrid algorithm that our algorithm has no redundant operations.

References