New Strategies for Computing the Transitive Closure of a Database Relation

Hongjun Lu †

Honeywell Inc.
Corporate Systems Development Division
Golden Valley, MN 55427

ABSTRACT

A large number of algorithms have been developed to compute the transitive closure of a database relation. This paper presents two new strategies that further reduce the data size dynamically during the computation and speed up the convergence to the least fixed point of the transitive closure relation. A hash-based algorithm that integrates these new strategies is then developed. The performance analysis indicates that the new algorithm outperforms other algorithms in most cases.

1. Introduction

Recursive query processing is one of the key problems in integrating database technology and artificial intelligence technology to develop *expert database systems*. Among the large family of recursive queries, a *transitive closure query*, a query whose processing requires the computation of the transitive closure of a database relation, is a relatively simple but very important class of recursive queries. They are important because (i) a large number of recursive queries can be expressed using transitive closures [Agra87, Rose86], (ii) most applications problems involving recursive queries which we can see now are actually transitive closure queries, and (iii) efficient processing of transitive closure queries will provide a sound base for solving more complicated recursive queries. It is thus not surprising that much effort has been devoted to the efficient computation of the transitive closures of database relations recently [Ioan86, Lu87, Rose86, Vald86]. There is even a tendency to extend relational algebra to include the operation of transitive closure in relational database management systems [Agra87].

After examining the available transitive closure algorithms that are summarized in Section 2, we feel that it is possible to further optimize the computation of the transitive closures of database relations. In Section 3, two new strategies are presented. In Section 4, an algorithm based on these new strategies and a modified hash join method is proposed, and its performance is compared with two previous algorithms. The last section discusses some possible extensions to the algorithm.

This research was supported in part by Rome Air Development Center contract F30602-85-C0216.

† Current Address : Department of Information Systems and Computer Science, National University of Singapore, Kent Ridge, Singapore 0511

2. Transitive Closure of a Database Relation

If $R_0(a,b)$ is a transitive database relation, its transitive closure $R = R_0^+$ is defined by

$$R = R_0^+ = \bigcup_{i \geq 1} R^i$$

where R^i denotes the i^{th} power of R_0: $R^1 = R_0$ and $R^n = R^{n-1} \circ R$ for $n > 1$. The composition operator $\circ$ on the two binary relations R and S is defined by

$$R \circ S = \{(x,z) \mid \exists y\, (x,y) \in R \wedge (y,z) \in S\}$$

Using relational algebra, this composition can be expressed as

$$R \circ S = \pi_{R.a, S.b}(R \bowtie_{R.b = S.a} S)$$

Graphically, relation R_0 can be represented as a directed graph $G(V,E)$, where a node $a \in V$ represents a domain value of $a \in \{R_0.A, R_0.B\}$, and a directed edge e in E, $a \rightarrow b$, represents a tuple (a,b) in the relation R_0. Then, a node pair (x,y) is in the transitive closure of R_0, R (or R_0^+) whenever there is a path of nonzero length from x to y. The longest path length, that is, the largest number of edges comprising a path, is sometimes referred to as the *depth* of the transitive closure. We will follow the same convention in our discussion.

More formally, the transitive closure of relation R_0 represents the derived relation R defined by the following Horn clauses:

$$R(x,y) :- R_0(x,y).$$

$$R(x,y) :- R(x,z), R_0(z,y).$$

2.1. Algorithms Computing Transitive Closure

In this subsection we are going to briefly summarize the algorithms proposed in the literature that compute the transitive closure of a database relation. These algorithms can be divided into two groups: *iterative* and *recursive* algorithms.

2.1.1. Iterative Algorithms

The iterative algorithms compute transitive closure R of a database relation R_0 by computing the least fixed point of the following equation:

$$R = R_0 \cup \pi_{R.b, R_0.a}(R \underset{R.b=R_0.a}{\bowtie} R_0)$$

A number of algorithms have been developed to implement this computation.

Naive algorithm

The most straightforward method, the *naive* algorithm, follows the semantics of the above least fixed point equation and uses the following iterative program:

```
old_R = $R_0$;
do {
  $R = old\_R \cup old\_R \circ R_0$;
  $\Delta R = R - old\_R$;
  $old\_R = R$;
} while ( $\Delta R \neq \emptyset$ )
```

Semi-naive algorithm

The naive algorithm is inefficient since it uses the whole result relation generated so far in each iteration to obtain more results and thus duplicates some effort in the computation. In fact, only tuples generated in the most recent iteration will introduce new tuples into the transitive closure. The following algorithm eliminates such duplication:

```
$R = R_0$;
$\Delta R = R_0$;
while ( $\Delta R \neq \emptyset$ ) {
   $\Delta R = \Delta R \circ R_0$;
   $\Delta R = \Delta R - R$;
   $R = R \cup \Delta R$;
}
```

Adopting well known terminology [Banc85], this algorithm is called the *semi-naive* algorithm.

Logarithmic algorithm

The semi-naive algorithm optimizes the computation of transitive closures of a relation by reducing the data size involved in the computation. Valduriez and Boral proposed another algorithm [Vald86] that optimizes the computation by reducing the number of iterations but handling larger data sets during each iteration. The algorithm is as follows:

```
$R = R_0$;
$\Delta R_0 = R_0$;
$\delta R_0 = R_0$;
while ($\Delta R \neq \emptyset$) {
   $\delta R = \delta R \circ \delta R$;
   $\Delta R = R \circ \delta R$;
   $R = R \cup \Delta R \cup \delta R$;
}
```

In this algorithm, after iteration j, the result relation R contains the tuples in $R_0, R_0^2, \cdots, R_0^{2^{j+1}-1}$, that is,

$$R = R_0 \cup R_0^2 \cdots \cup R_0^{2^{j+1}-1}$$

Therefore, if the depth of the transitive closure is ρ, only $lg(\rho+1) - 1$ iterations are needed to complete the computation. In iteration j, two joins are computed. The first computes the join of R^j and R^j, and the second joins the result of the first join with the result relation obtained in the last iteration. That is, for each iteration, more tuples are processed, and more result tuples are generated than the naive and semi-naive algorithms. The computation converges to the least fixed pointer faster.

Smart algorithms

Ioannidis recently proposed a new set of algorithms, *smart* algorithms, to compute the transitive closure of a relation [Ioan86]. A frame work of optimizing the computation along the same direction as the logarithmic algorithm was provided. According to the smart algorithms, the transitive closure of relation R_0 is expressed as

$$R^+ = \prod_{l=0}^{\infty}\left(\sum_{k=0}^{m-1} R_0^{k \cdot m^l}\right)$$

With a different m value, different algorithms can be obtained. The logarithmic algorithm is actually the special case of m=2.

$$R^+ = (1+R_0)(1+R_0^2)(1+R_0^4)\cdots$$

2.1.2. Recursive Algorithm

Lu *et al.* adapted the Warren's algorithm, which is used for computing the transitive closure of a binary relation represented by a boolean matrix, to compute the transitive closure of a database relation [Lu87]. This algorithm is recursive in nature. When a tuple t in R_0 is processed, all result tuples derivable from t are generated. For implementation reasons, the algorithm sorts the relation first and then processes it in two passes:

```
T = $R_0$ sorted on attributes <A, B>;
foreach ( $t \in T$ and $t.A >_D t.B$ ) do
  begin
    findall t' in T where t.B = t'.A;
    insert { (t.A, t'.B) } into T;
  end;
foreach ( $t \in T$ and $t.A <_D t.B$ ) do
  begin
    findall t' in T where t.B = t'.A;
    insert { (t.A, t'.B) } into T;
  end;
```

where $>_D$ represents the partial ordering on domain D.

2.1.3. Use of Join Indices

In order to reduce the data in the computation of transitive closures, Valduriez and Boral also suggested applying the logarithmic algorithm to a data structure called join indices instead of the relation itself [Vald86]. A join index on two relations, $R(A,B)$ and $S(A,B)$, is defined as the set of

$$JI = (r_i, s_j \mid r_i.A = s_j.B)$$

where r_i and s_j represent the tuples of R and S. If the join selectivity between two relations is low (that is, the number of tuples in the result of a join between R and S is far less than the product of the number of tuples in R and S), the size of the join index will be small. Their analysis indicated that both the semi-naive and logarithmic algorithms perform better when they are applied to the join indices than when applied to the original relations.

† The non-iterative algorithm does not require explicitly the join operations. However, the processing of each tuple can be viewed as a join of the tuple and a subrelation of R.

2.1.4. Discussion

We have briefly summarized the major algorithms proposed in the literature that compute the transitive closure of a database relation. Except the naive algorithm, which is apparently inefficient, other iterative algorithms and the recursive algorithm have their own merits and deficiencies. The real performance will depend on the application and the characteristics of the relation for which the transitive closure is computed [Ioan86, Lu87, Vald86].

3. New Strategies Optimizing the Computation

In this section, we are going to propose two new strategies that further optimize the computation of the transitive closure of a database relation. We first assume that the relation we are dealing with is so large that it is impossible to hold all its tuples in main memory. In this case the computation of transitive closure, no matter which algorithm is used, requires a large number of join, union and set difference operations on very large relations. † Partitioning a very large relation into smaller disjoint partitions has been proved a reasonable way to dramatically reduce the costs of join operation on large relations [DeWi84]. Both the analysis of the recursive algorithm [Lu87] and the logarithmic algorithm [Vald86] are based on the hash join method. We assume that the same technique is used in our discussion.

3.1. Strategy 1: Reduce the Size of R_0

Compared to the naive algorithm, the semi-naive algorithm focus on eliminating the duplication of computation by only using the newly generated tuples as one of the source relations of the join in the next iteration. However, none of the previous algorithms tried to reduce the size of another source relation in the join operation, relation R_0. Since relation R_0 is used in each iteration, its size perhaps has more influence on the performance of the transitive closure algorithms.

Our first optimization strategy is to eliminate dynamically those tuples from relation R_0 that will not generate tuples in the result relation in the later iterations. The next example is used to explain the strategy. Relation R_0 consists of 13 tuples. Figure 1 is a graph which represents R_0.

Table 1 shows the tuples generated during computing R_0^+. For the semi-naive algorithm, the first iteration joins R_0 with R_0 and generates $\Delta R_1 = R_0 \bowtie R_0$, which consists of 12 tuples. Traditionally, the second iteration will join ΔR_1 with R_0 again to generate $\Delta R_2 = \Delta R_1 \bowtie R_0$. However, if we examine the join process, we can find that some tuples in R_0 will never introduce new tuples. These tuples, in the column of R_0 above the dotted line, can actually be removed from R_0 without affecting the final result. A new relation $R_0{}^1$ formed in this way can be used in the second iteration to compute ΔR_2. In this example, $R_0{}^1$ consists of only 6 tuples, less than 50 percent of R_0.

Figure 2 lists algorithm REDUCE, an algorithmic description of the suggested strategy for reducing the size of relation R_0. The notation used is similar to that used in the semi-naive algorithm: two relations to be joined in iteration i are ΔR_i and R_0^i. ΔR_i contains new tuples in the transitive closure generated in the $(i-1)^{th}$ iteration. Relation $R_0^0 = R_0$, and R_0^i is reduced to R_0^{i+1}, which is to be used in the next iteration to join with ΔR_{i+1}. Note that algorithm REDUCE as described above is for general cases. For a particular algorithm, for example, the semi-naive algorithm, the removal of tuples from ΔR_i is only needed for the first iteration of join R_0 and R_0: for the semi-naive algorithm, ΔR_i only contains newly generated tuples which are not in R_0.

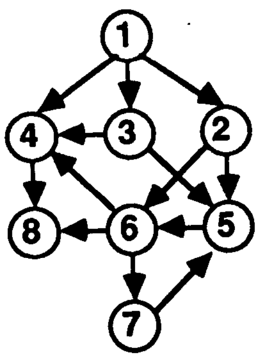

Figure 1: Graph of Relation R_0.

Iteration 0		Iteration 1		Iteration 2	
R_0	R_0	ΔR_1	R_0^1	ΔR_2	R_0^2
(1, 2)	(1, 2)	(1, 5)	(4, 8)	(1, 7)	(4, 8)
(1, 3)	(1, 3)	(1, 6)	(5, 6)	(7, 4)	(5, 6)
(1, 4)	(1, 4)	(1, 8)	(6, 4)	(3, 7)	(6, 4)
(3, 4)	(2, 5)	(3, 8)	(6, 7)	(7, 7)	(6, 7)
(6, 4)	(2, 6)	(3, 6)	(6, 8)	(7, 8)	(6, 8)
(2, 5)	(3, 4)	(7, 6)	(7, 5)	(2, 8)	(7, 5)
(3, 5)	(3, 5)	(2, 4)		(5, 5)	

(7, 5)	(4, 8)	(2, 7)		(5, 6)	
(2, 6)	(5, 6)	(2, 8)			
(5, 6)	(6, 4)	(5, 4)			
(6, 7)	(6, 7)	(5, 7)			
(4, 8)	(6, 8)	(5, 8)			
(6, 8)	(7, 5)	(6, 5)			

Table 1: Computing R_0^+.

```
Algorithm REDUCE:
Input : Two intermediate relations $\Delta R_i$ and $R_0^i$
Output : Relation $R_0^{i+1}$

begin
  repeat
    foreach tuple $t \in R_0^i$ do
      begin
        if $\Delta R_i \bowtie t = \varnothing$
        then begin
                remove $t$ from $R_0^i$;
                if $t \in \Delta R_i$
                then remove $t$ from $\Delta R_i$;
             end;
      end;
  until no tuple can be removed from $R_0^i$;
  $R_0{}^{i+1}$ := $R_0^i$;
end;
```

Figure 2: Algorithm Reducing the Size of R0.

Graphically, removing tuples as described in the algorithm is the process of removing outgoing edges from nodes satisfying the following conditions: (i) there is no incoming edge to the node, and (ii) all outgoing edges are already inserted to the relation. The second condition is automatically satisfied because the original relation R_0 is copied into the result. Since there is no incoming edge to the node, no more paths can be generated via the node, and its removal from the graph will not lose results. In the above example, node 1 has no incoming edges; after the edges started from it are inserted to the result relation, it can be removed along with those edges. This removal of node 1 further causes the removal of nodes 2 and 3, since only incoming edges for nodes 2 and 3 are from node 1.

For large database relations, it will be very expensive if algorithm REDUCE is implemented as it is described in Figure 2. In the next section, one possible implementation is described which modifies the hash join method to dynamically reduce the size of R_0 without heavy overhead. Another point we would like to make is that this strategy has some flavor of using join indices to compute the transitive closure [Vald86]: only those tuples which are *joinable* are kept for computation. However, join indices are static data structures and do not change for different iterations of the computation. In our algorithm, size reduction is dynamically performed. We have the benefit of reducing the data size without the disadvantage associated with join indices: the costs of generating the join indices and maintaining them in a database; the difficulty of determining which relations and on which attributes the join indices should be maintained; and the complexity to determine whether it is beneficial to use the join indices.

3.2. Strategy 2: Speed Up the Convergence

The number of iterations needed to complete the transitive closure computation is another source of optimization. The logarithmic algorithm and smart algorithms outperform the semi-naive algorithm since they generate more tuples in one iteration and fewer iterations are needed. Intuitively, the source relations are only read from the disks once in one iteration. The more tuples generated in one iteration, the fewer number of iterations needed to complete the computation. Thus, one of the major processing costs, disk I/Os for reading in the source relation, is reduced. The CPU cost, such as rehashing, if hash join is used, is also reduced partly. The savings gives the logarithmic and smart algorithm better performance [Vald86, Iann87].

The recursive algorithm is an extreme along this direction: when a tuple is processed, all tuples derivable from this tuple are generated. The performance of the algorithm is irrelevant to the maximum path length of the transitive closure of the relation. If there are some very long paths in the transitive closure, this algorithm will outperform the iterative algorithms. The limitation of this algorithm is that, in order to find all tuples derivable from a tuple, the processing has the flavor of the depth-first search. In cases where the size of memory is much smaller than the relation size, a large amount of disk access is required, which leads to bad performance [Lu87].

The strategy suggested here combines the iterative methods with the recursive algorithm. For each pair of buckets which can be held in main memory, all tuples in the transitive closure derivable from them are generated. These tuples are output either to the corresponding buckets for further processing or to the final result relation.

Algorithm PROCESSING in Figure 3 describes the algorithm of processing the i^{th} bucket pair in the k^{th} iteration using the strategy. ΔR_i^{k-1} contains the tuples generated in iteration k-1 and is hashed on the second attribute. ${R_0}_i^k$ is the corresponding bucket partitioned on the first attribute. R_i is the i^{th} bucket of the result relation R, the transitive closure of R_0. Function GetBucketNo() returns the bucket number a tuple belongs to when hashing on the second attribute. The algorithm works as follows: for each tuple $t(a,b)$ in ΔR_i^{k-1}, it finds all matching tuples from ${R_0}_i^k$. New tuples are formed and hashed on the second attribute to find the buckets to which the tuples belong. The tuples falling to the current bucket are used to further probe the hash table.

```
Algorithm PROCESSING:
Input  :  A pair of buckets, $\Delta R_i^{k-1}$, ${R_0}_i^a$
Output :  Tuples in the transitive closure of $R_0$
          (inserted into corresponding buckets)

begin
   foreach tuple $t$ in $\Delta R_i^k$ do
probe:
      if there is a match tuple $t'$ in ${R_0}_i^k$ with $t.B = t'.A$
      then begin
            form a new tuple $newt(t.A, t'.B)$;
            j = GetBucketNo($t'.B$);
            if (j > i)
            then output $newt$ to $\Delta R_j^{k-1}$;
            if (j < i)
            then output $newt$ to $\Delta R_j^k$;
            if   (j = i)
            then if ($t.A \neq t'.B$)
                then goto probe;
                else output $newt$ into $R_i$;
          end;
end;
```

Figure 3: Algorithm PROCESSING.

Tuples of other buckets are output to the corresponding buckets. They are either processed in the same iteration (if the bucket has not been processed yet), or processed in the next iteration. For each tuple, the processing will terminate when cyclic data (a tuple $t(a,a)$ is obtained) is encountered, or no more matching tuples can be found in ${R_0}_i^k$.

We use a simple example to explain the algorithm. Relation R_0 shown in Figure 4 consists of 8 tuples. They are partitioned into two pairs of buckets, $(R^b{}_{0_1}, R^a{}_{0_1})$ and $(R^b{}_{0_2}, R^a{}_{0_2})$, on attribute b and a, respectively, because of the limitation of memory size. A tuple $t(a,b) \in R^b{}_{0_1}$ iff $hash(t.b)$ in {1, 2, 3} and $t(a,b) \in R^b{}_{0_2}$ iff $hash(t.b)$ in {4, 5, 6}. Partitions $R^a{}_{0_1}$ and $R^a{}_{0_2}$ are formed in a similar way.

Table 2 shows the tuples generated during computing R_0^+ using the algorithm. The computation starts with the first pair of buckets, $\Delta R_1^0 = R^b{}_{0_1}$ and $R^0{}_{0_1} = R^a{}_{0_1}$. Algorithm PROCESSING is applied and the result tuples are hashed on the second attribute. Those tuples with hash values in {4, 5, 6} (five of them in this example) are appended to the bucket ΔR_2^0 (as shown in the figure under the dotted line). Other tuples (in this case, three) are output as the result. The second pair of buckets is processed in a similar way. The difference is that the tuples generated with the hash value of the second attribute in {1, 2, 3} are used to form ΔR_1^1, which is used in the next iteration. †

This strategy can be explained intuitively with the graphic representation of R_0 as follows: The hashing technique partitions the directed graph, G_0 into a number of subgraphs ${G_0}_i$. An edge e: $a \xrightarrow{e} b$ is in subgraph ${G_0}_i$ iff b is in bucket i. For each edge $e \in {G_0}_i$ ($a \xrightarrow{e} b$), algorithm PROCESSING finds all paths that start from node a and are contained in subgraph ${G_0}_i$. If there is a path leading to a node c in another subgraph, ${G_0}_j$, the output of

† In the example we did not show the elimination of duplicates: duplicates in the result tuples are eliminated before the next iteration, as when using the semi-naive algorithm.

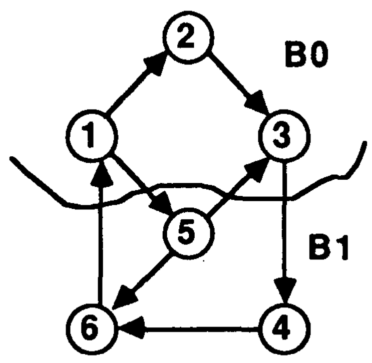

Figure 4: An Example Relation R_0.

			Iteration 1		
ΔR_i^1	$R_{0_i}^1$	R_i^1	ΔR_i^2	$R_{0_i}^2$	R_i^2
(6, 1)	(1, 2)	(6, 2)	(3, 1)	(1, 2)	(3, 2)
(1, 2)	(1, 5)	(6, 3)	(4, 1)	(1, 5)	(3, 3)
(2, 3)	(2, 3)	(1, 3)	(5, 1)	(2, 3)	(4, 2)
(5, 3)	(3, 4)		(2, 1)	(3, 4)	(4, 3)
					(4, 4)
					(5, 2)
					(2, 2)
(3, 4)	(4, 6)	(3, 6)	------	(4, 6)	
(1, 5)	(5, 3)	(1, 6)	(3, 5)	(5, 3)	
(4, 6)	(5, 6)	(1, 1)	(4, 5)	(6, 1)	
(5, 6)	(6, 1)	(6, 6)	(5, 5)		
------		(2, 6)	(2, 5)		
(6, 4)					
(6, 5)					
(1, 4)					
(2, 4)					
(5, 4)					

Table 2: An Example of Using Strategy 2.

a tuple (a,c) to bucket ΔR_j^p during the processing can be viewed as inserting a node a and an edge $a \rightarrow c$ in subgraph G_{0_j}. Therefore, any path starting from node a in subgraph G_{0_j} and ending with another node b in subgraph G_{0_j} can be internally found in subgraph G_{0_j} later on.

The effectiveness of this strategy is clearly shown by the example in Figure 4. The longest path in the transitive closure includes five edges ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 1$), which requires five iterations for the semi-naive algorithms and three iterations for the logarithmic algorithm. However, only two iterations are needed using our strategy.

From the example, we can also see some savings other than the reduction of the number of iterations. In the previous iterative algorithms, new tuples generated during computation have to be read in at least once to join with the original relation. In our strategy, the result tuples corresponding to the paths which do not cross the border of subgraphs are not read in again. In the example, among 23 tuples generated in the transitive closure (excluding the original tuples in R_0), only 12 tuples are written out and then reread in for later processing.

4. Algorithm HYBRIDTC

In this section, we describe a hash-based transitive closure algorithm. It integrates the strategies described in the last section. Since this algorithm combines the merits of both iterative and recursive methods, we name it algorithm HYBRIDTC (a *hybrid* transitive closure algorithm).

4.1. The Algorithm

```
Algorithm HYBRIDTC;
Input : relation $R_0$
Output: relation $R$, the transitive closure of relation $R_0$

begin
    partition $R_0$ on $R_0.A$ and $R_0.B$ into
          buckets $R_{0_i}^a$ and $R_{0_i}^b$ ($1 \le i \le N$);
    for i := 1 to N do begin
      $\Delta R_i^1 := R_{0_i}^b$;
      $R_{0_i}^1 := R_{0_i}^a$;
    end;
    k := 0;
    repeat
        k := k + 1;
        for i := 1 to N do
            if ($\Delta R_i^k \neq \emptyset$) and ($R_{0_i}^k \neq \emptyset$)
            then ProcessingBucket (i, k, $\Delta R_i^k$, $R_{0_i}^k$);
            else $\Delta R_i^k := \emptyset$;
        for i := 1 to N do begin
            $\Delta R_i^k := \Delta R_i^k - R_i^{k-1}$;
    until all $\Delta R_i^k$s are empty;
    $R := \bigcup_{1 \le i \le N} R_i$;
end.
```

Figure 5: Algorithm HYDRIDTC.

The algorithm is shown in Figure 5. Relation R_0 is partitioned into two sets of buckets on attribute $R_0.A$ and $R_0.B$ as in traditional hash joins. These two set of buckets are denoted by $R_{0_i}^b$ and $R_{0_i}^a$ ($1 \le i \le N$), respectively. We will use subscripts to denote the bucket number and superscripts to denote the iteration number. Let ΔR_i^k contain the new tuples in the transitive closure that belong to bucket i (hashed on attribute B) generated during the $(k-1)^{th}$ iteration, and $R_{0_i^k}$ be the reduced bucket i of R_0 after $(k-1)$ iterations. The bucket pair processed in the k^{th} iteration is ΔR_i^k and $R_{0_i}^k$, where $\Delta R_i^1 = R_{0_i}^b$, and $R_{0_i}^1 = R_{0_i}^a$.

After the relation is partitioned, the ΔRs are initialized to be the corresponding set of buckets. The processing of bucket pairs proceeds iteratively until all ΔR_i^ks are empty for the k^{th} iteration. Since ΔR_i^k contains the most recently generated tuples, and $R_{0_i}^k$ is also reduced during each iteration, procedure *Processing-Bucket* is only called when both of them are nonempty. During the processing of bucket pair ΔR_i^k and $R_{0_i}^k$, some result tuples are inserted into R_i, and others are inserted to other buckets ΔR_j, ($j \neq i$), as described in algorithm PROCESSING. After each iteration k, duplicates are eliminated from the ΔR_i^{k+1}s which are going to be used in the next iteration.

The union and duplicate elimination procedures are the same as any transitive closure algorithms, and we are not going to discuss them here. Figure 6 and Figure 7 give one possible implementation of the procedures *ProcessingBucket* and *ProcessingTuple*. In this implementation, a hash table is constructed for $R_{0_i^k}$ as in the traditional hash join algorithms. However, one

```
procedure ProcessingBucket
        ( bucketno, iteration : integer;
          deltabucket, bucketR0 : buckets );
begin
  BuildHashTable(bucketR0);
  foreach tuple in deltabucket do
    ProcessingTuple(bucketno, iteration, tuple);
  foreach tuple in the hash table do
    if   tuple.mark
    then  OutputBucketR0 (tuple, bucketno, iteration+1);
end;
```

Figure 6: Procedures *ProcessingBucket*.

extra field "*mark*" is added to the hash table entry. It is used to mark the tuples actually participating in the join. Procedure *ProcessingTuple* is called for each tuple in *deltabucket* (ΔR_i^k). After all tuples have been processed, only those marked tuples are written back by calling procedure OutputBucketR0 to form $R_{0_i}^{k+1}$.

```
procedure ProcessingTuple
      ( bucketno, iteration: integer; inputuple : TupleType );

var  currenttuple, matchtuple, newtuple : TupleType;
     newbucketno : integer;
begin
  PushStack(inputtuple);
  while (NOT EmptyStack) do
    begin
      currenttuple := PopStack;
      if (currenttuple.a <> currenttuple.b)
      then begin
        matchtuple := LookUp(currenttuple);
        foreach matchtuple do
          begin
            if (NOT matchtuple.mark)
            then matchtuple.mark := true;
            newtuple := FormTuple ( currenttuple.a,
                                    matchtuple.b);
            newbucketno := GetBucketNo(newtuple);
            if (newbucketno = bucketno)
            then PushStack(newtuple);
            if (newbucketno < bucketno)
            then OutputDelta(newtuple,
                     newbucketno, iteration+1);
            if (newbucketno > bucketno)
            then OutputDelta(newtuple,
                     newbucketno, iteration);
          end;
      end;
      OutputResult(bucketno, currenttuple);
    end;
end; (* procedure ProcessingTuple *)
```

Figure 7: Procedure of Processing a Tuple in ΔR_i^k.

Procedure *ProcessingTuple* implements strategy 2 using a stack of tuples. *PushStack*, *PopStack*, and *EmptyStack* are procedures and functions manipulating the stack. The tuple on the top of the stack is used to look up the hash table to find matches. Those matching tuples can be divided into three categories according to the bucket it belongs to. The bucket number of a tuple is returned by function GetBucketNo. The tuples of other buckets are inserted to ΔR_j buckets by procedure OutputDelta. The tuples of current processing buckets are pushed onto the stack for later processing. This process continues until the stack is empty.

The advantage of using a stack is its simplicity. Another advantage, perhaps a more important one, is ease of memory management. If there is a large number of tuples derived from some particular tuple in the bucket which leads to a full stack, we can just write part of the bottom of the stack on the disk and reread it back in to free memory space later on for continuing the process. Thus, algorithm HYBRIDTC does not introduce new issues in memory management. Techniques of partitioning a relation into buckets and of handling overflow buckets developed in hash join methods can be directly used.

Now, we prove the following Lemma:

Lemma: *Algorithm HYBRIDTC correctly computes the transitive closure of a database relation.*

Proof: The proof of the Lemma consists of two parts. First, we have already explained in Section 2 that the removal of unmarked tuples, the tuples not participating in the join in the current iteration, will not lead to loss of the result tuples. Second, we prove that the algorithm will find all tuples in the transitive closure. In other words, the algorithm can find all paths in graph G_0 if relation R_0 is represented by G_0. Let p be a path of graph G_0. It is obvious that, if all nodes on path p are contained in one subgraph of G_0, the path can be found by the algorithm when the corresponding buckets are processed. It is more likely that paths cross over the border of subgraphs. Let e, $(a \overset{\bullet}{\rightarrow} b) \in G_0$, be an edge, and the end nodes of e be a and b, and they are in two different subgraphs, G_{0_i} and G_{0_j} respectively. Then tuple (a,b) is in bucket j. During processing of bucket j, all paths of p starting from b and ending at some nodes y_j in G_{0_j} can be found, and a set of tuples $\{ (b, y_1), \ldots , (b, y_j), \ldots \}$ is generated. If there are some paths starting from some node x_i and ending at node a, the processing of bucket i will not only generate a set of tuples $\{x_i, a\}$, but also generate a set of tuples $\{x_i, b\}$. They are inserted into bucket j. Thus, in the next iteration of processing bucket j, all paths starting from node x_i and ending at node y_j can be found. The proof can be extended to the paths across any number of subgraphs. □

4.2. Performance Comparisons

Qualitatively, algorithm HYBRIDTC is expected to improve performance in the following ways:

(1) *Reduce the number of iterations.*

For the semi-naive and logarithmic algorithms, only paths with certain lengths can be found in each iteration. The number of iterations needed to complete the computation is determined by the depth of the transitive closure, that is, the longest path. For algorithm HYBRIDTC, paths contained in a subgraph can be generated in a single iteration no matter how long it is. Furthermore, the later processed buckets make use of the new tuples generated by the buckets which have been processed in the same iteration. As a result, the number of iterations needed largely depends on how the relation is partitioned and is usually less than the depth of the transitive closure. The reduction in the number of iterations at least reduces the disk I/O needed to read in R_0 and CPU time for constructing the hash tables.

(2) *Reduce the number of disk I/Os needed to read in the delta relations.*

For both the semi-naive and logarithmic algorithms, the result tuples generated in one iteration have to be written to the disk and read in again in the next iteration. However, in algorithm HYBRIDTC, the tuples generated in one iteration need to be read in again only if they belong to other buckets. Again, the extent of this savings largely depends on the data distribution and the partitions.

(3) *Reduce the size of the source relation.*

As explained in Section 2, the source relation used to compute the transitive closure is dynamically reduced during processing, compared to the constant size in the semi-naive algorithm and no optimization in the logarithmic algorithm.

Any quantitative analysis of algorithm HYBRIDTC is difficult, since the performance will vary dramatically with different data characteristics and the partitioning. In order to validate our qualitative analysis above, we made some comparisons between the performance of the semi-naive algorithm, the logarithmic algorithm, and algorithm HYBRIDTC as follows:

(1) The data model proposed by Bancilhon and Ramakrishnan [Banc86] is used. We examined two simple cases, lists and trees having fanout 2.

(2) We use the number of tuples read in during the computation as the performance measure for the comparison. This number roughly reflects the total costs of the computation. The larger the number is, the more disk I/O cost and CPU cost for constructing the hash tables. Furthermore, we assume that duplication elimination costs are the same for all three algorithms, and they are not taken into account.

(3) Some of the implementation details are ignored. For example, for the semi-naive algorithm and the logarithmic algorithm, we only calculate the total number of tuples of two relations joined in each relation. This number is therefore independent of the memory size and the number of hash buckets. We actually assume that the pipeline method is used to reduce the number of disk I/Os [Lu87]. That is, each tuple in the transitive closure only counts once: no separate partition phase is assumed.

With the above assumptions, the total number of tuples for the semi-naive and the logarithmic algorithms are calculated as follows:

For the semi-naive algorithm, h iterations are needed to generate all tuples in the transitive closure. One more iteration is actually completed, resulting in the termination of the computation. During each iteration, there is only one join. The total number of tuples participating in the join operations is:

$$N_{semi-naive} = ||R|| + (h+1)||R_0||$$

The number of iterations needed in the logarithmic algorithm, k, is determined by $k = lg(h+1) - 1$. For each iteration i, there are two joins: the join of R^i with R^i, and the join of the result tuples in the transitive closure so far, which is $\sum_{j=1}^{2^i-1} R^j$, with the newly generated relation R^{2^i}. The total number of tuples participating in the computation is:

$$N_{logarithmic} = \sum_{i=1}^{k}(2^*R_0^i + \sum_{j=0}^{2^i} R_0^j)$$

The number of tuples read in algorithm HYBRIDTC is obtained by simulation: a program was coded to implement the algorithm in memory. A random number generator was used to assign bucket numbers for tuples. The corresponding buckets were then joined iteratively to compute the transitive closure. When each bucket pair was processed, the number of tuples in the buckets was counted. The total number of tuples read in could thus be obtained. In the simulation, we used a small bucket size (typically each bucket contains 10 tuples). Therefore the simulation actually does not favor algorithm HYBRIDTC.

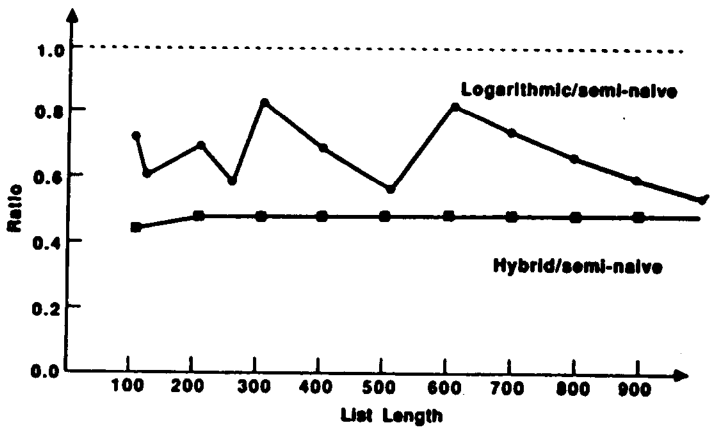

Figure 8: Performance Comparsion 1 (R_0: List).

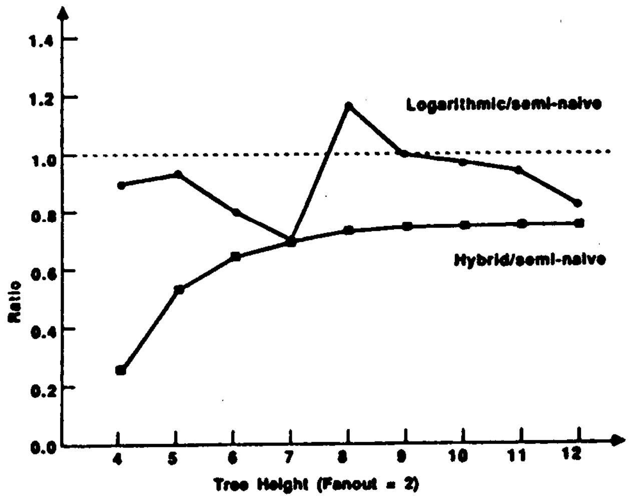

Figure 9: Performance Comparsion 2 (R_0: Tree).

The result of this comparison is shown in Figures 8 and 9. The lengths of the lists vary from 100 to 1024. The tree depth varies from 4 to 12. The comparison uses the number of tuples in the semi-naive algorithm as a reference. The ratio of logarithmic/semi-naive and hybrid/semi-naive are computed. The results in the figures show that algorithm HYBRIDTC consistently outperforms the other two algorithms. For lists, the ratio hybrid/semi-naive is about 50 percent. However, the ratio of logarithmic/semi-naive is about 60 to 70 percent. This result is expected as we discussed above.

In both figures, the ratio of logarithmic to semi-naive is not monotonic. Sometimes, the semi-naive algorithm even outperform the logarithmic algorithm. This happens when the depth is just larger than 2^k. This is also observed by Ioannidis [Ioan86]. The explanation is that the number of iterations of the logarithmic algorithm is determined by the depth. When the depth increases to past 2^k, the number of iterations increases by 1. That is, another iteration is required to complete the computation to find just a few more tuples. That is one disadvantage of the logarithmic algorithm.

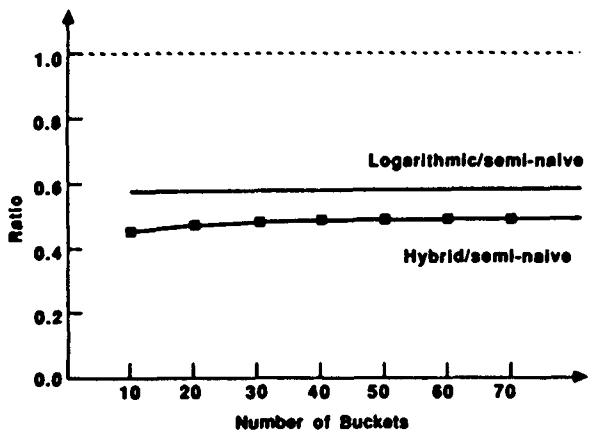

Figure 10: Performance vs. Number of Buckets (R_0: List).

We did not compare the performance of algorithm HYBRIDTC with the recursive algorithm. Its performance becomes much worse than the logarithmic algorithm when the memory size is small, compared with the relation size [Lu87]. However, algorithm HYBRIDTC still performs better than the other two algorithms, even in this case. Figure 10 illustrates the number of disk I/O tuples with the different number of buckets into which the relation is partitioned. When we increase the number of buckets, which simulates smaller and smaller bucket size, the number of disk I/O tuples also increases. However, it is still less than what needed in the other two algorithms.

5. Conclusions

We have discussed two strategies which optimize the computation of the transitive closure of a database relation. We also presented a hash-based algorithm that integrates these two strategies together. The algorithm is easy to implement in real systems by modifying the traditional hash join methods. A simple performance analysis was conducted, and the results indicate that the new algorithm does outperform previous algorithms. This performance analysis is far from complete. However, it does provide the evidence that our new strategies in optimization are in the right direction. Further detailed implementation in relational database systems and performance analysis is one of the possible projects for future work.

Besides better performance, the algorithm has some other advantages. For example, the algorithm is easy to extend to become a distributed or parallel algorithm. In algorithm HYBRIDTC, there is no inherent sequence among the iterations. For other algorithms, the result of an iteration is used as the input of the next iteration. In the logarithmic algorithm the second join in each iteration can only be started after the first join finishes. For the distributed version of algorithm HYBRIDTC, each processor or node can work on one or more pairs of buckets. The tuples generated at one processor are either processed locally or sent to other processors. The only synchronization needed is the final termination of the whole computation.

This algorithm can be further optimized along the directions proposed. One possibility is as follows: the new tuples generated are not only hashed on the second attribute and inserted into the corresponding buckets, but also hashed on the first attribute and inserted into the second relation in the join ($R_{0_i}^{k}$). Thus, more tuples can be generated in each iteration, and performance improvement can be expected. However, it is somewhat difficult to implement in real system since the size of $R_{0_i}^{k}$ will change during processing. Some sophisticated memory management strategy and bucket overflow techniques have to be developed.

Algorithm HYBRIDTC is a basic algorithm for computing the simple transitive closure of a relational database. Interesting future work is to use it as a base for extending a relational database management system to include transitive closure as one basic operation. To achieve this, the algorithm should be further augmented so that more complicated transitive closure queries can be processed efficiently [Agra87].

Acknowledgement

The author wishes to thank Guy Lohmen who provided with suggestions which improved this paper.

References

[Agra87] Agrawal, R., "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries," in *Proceedings of the Third International Conference on Data Engineering*, Los Angeles, CA., February 3-5, 1987.

[Banc85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations," in *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, February 1985.

[Banc86] Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies," in *Proceedings of the 1986 ACM-SIGMOD Conference*, Washington, D.C., May 1986.

[DeWi84] DeWitt, D. J., Katz, R.H., Olken, F. Shapiro, L. D., Stonebraker, M. R., and Wood, D., "Implementation Techniques for Main Memory Database Systems," in *Proceedings of the 1984 ACM-SIGMOD Conference*, Boston, MA, June 1984.

[Ioan86] Ioannidis, Y. E., "On the Computation of the Transitive Closure of Relational Operations," in *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, August 1986.

[Lu87] Lu, H., Mikkilineni, K. , and Richardson, J. P., "Design and Evaluation of Algorithms to compute the Transitive Closure of a Database Relation," in *Proceedings of the Third International Conference on Data Engineering*, Los Angeles, CA., February 3-5, 1987.

[Rose86] Rosenthal, A., Heiler, S., Dayal, U., and Manola, F., "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," in *Proceedings of the 1986 ACM-SIGMOD Conference*, Washington, D.C., May 1986.

[Vald86] Valduriez, P., and Boral, H., "Evaluation of Recursive Queries Using Join Indices," in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.