A Hash-Based Approach for Computing the Transitive Closure of Database Relations

Farshad Fotouhi, Andrew Johnson, S. P. Rana

Computer Science Department
Wayne State University
Detroit, Michigan 48202
USA

Abstract

Recursive query processing is one of the key problems in integrating database and artificial intelligence technologies. Among the classes of recursive queries, transitive closure queries are the simplest, but most important. Here, we present an efficient hash-based algorithm for computing the transitive closure of database relations. Hashing is used to reduce the data size dynamically. The original data is partitioned once and the partitioning is maintained throughout the computation. Partitions are used in the computation instead of the entire relation. As the new tuples are generated after each iteration of the algorithm, these are placed in the appropriate partitions. We have shown the performance improvement of the proposed algorithm over the existing methods for a wide range of memory sizes, relation depths, and hash selectivity factors.

1. Introduction

Recursive query processing is an important part of knowledge-base systems. Jagadish and Rakesh[4] showed that any linearly recursive query can be expressed as a transitive closure, preceded and/or followed by relational algebra operations. Therefore efficient computation of the transitive closure is of key importance in evaluating recursive queries.

The transitive closure of a binary relation Ro with attributes A and B can be defined as:

$$Ro^{+} = \bigcup_{i \geq 1} R^{i} \text{ where } R^{i} = R^{i-1} \bullet R^{0} \text{ where } R^{0} = Ro$$

and S • T, the composition of S and T, is defined as:

$$S \bullet T = \{(x,z) \mid \exists y\ (x,y) \in S \text{ and } (y,z) \in T\}$$

Relation R can be represented as a directed graph which we refer to as a *relation graph.* The tuples correspond to edges in the graph and the unique attribute values correspond to the nodes. A tuple (x,y) in relation Ro becomes an edge from node x to node y in the directed graph. A tuple (x,y) in Ro^{+} means there is a non-zero length path from node x to node y. The longest path length in the graph (the depth of the transitive closure) is defined as:

$$p = \max_{\substack{x,y \\ paths(x,y) \neq 0}} \{\min_{q \in paths} [length(q)]\}$$

Several algorithms have been presented in the past to efficiently process the transitive closure of a relation[1,3,7,8]. Hashing techniques such as hybrid-hash[2] have been used in order to perform the composition, union and difference operations in many of the existing algorithms. In each iteration of these algorithms the entire original relation as well as the result of the previous step is hashed, and the desired operation is then performed. Here, we present a hash-based algorithm for computing the transitive closure of database relations. In the proposed algorithm, the original relation, say Ro(A,B), is first partitioned in two different ways. One partitioning is based on the A attribute values, and the other is based on its B attribute values. Subsequently, in each iteration of the algorithm, pairs of these partitions are joined, and the new tuples produced are placed in their appropriate partitions. This results in significant improvement over the existing methods because:

1) Rehashing is not required in each iteration.

2) Partitions begin to drop out as the computation proceeds (they will no longer be able to add new tuples to the result). This implies less and less of Ro will have to be read in during the join operation, and less and less of the result relation will have to be read in for duplicate removal. A similar improvement was suggested by Lu[6]. to reduce the number of tuples of Ro participating in an iteration. We have extended this so that partitions of Ro as well as partitions of the result relation drop out as the computation proceeds.

3) The algorithm is easily extendible to a parallel implementation. This is explored further by Johnson[5].

In section 2, we present existing algorithms for computing the transitive closure of database relations. Section 3 presents two new hash based algorithm. Section 4 gives the cost model for these algorithms, followed by a comparative analysis in section 5. We show the performance improvement of the proposed algorithms over the existing methods for a wide range of memory sizes, relation depth, and hash selectivity factor (i.e., the percentage of tuples dropping out in each iteration of the processing).

2. Existing Transitive Closure Algorithms

In this section, we describe major existing algorithms for computing the transitive closure of a database relation. Transitive closure algorithms are best classified into iterative and recursive algorithms. We consider the Semi-Naive and Logarithmic iterative algorithms, and Warren's recursive algorithm as representatives of each class.

An acyclic relation is often assumed in discussing transitive closure algorithms. This is to avoid the costly duplicate removal operation. This may cause an algorithm to compute the longest path between every pair of nodes, rather than the shortest path. For example, in Figure 1 without duplicate removal, the path a->e will be enumerated twice (viz. a->b->c->d->e, and a->d->e) requiring an extra iteration, and also extra work in each iteration. While duplicate removal is a costly process, it can cause a reduction in the number of iterations as well as the processing time within each iteration. For these reasons we relax the assumption of acyclicity.

Ro (A,B)

a	b
a	d
b	c
c	d
d	e

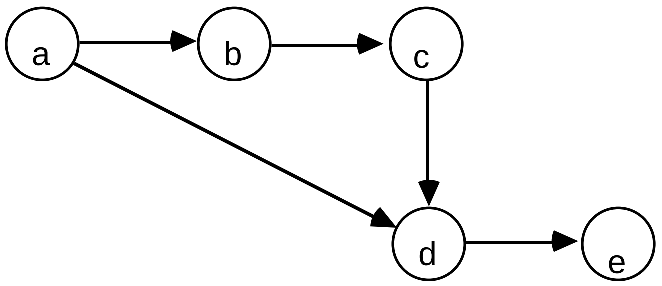

Figure 1

Relation Ro and its Corresponding Graph

In what follows, we use Ro(A,B) to denote the source relation, and T(A,B) to denote the transitive closure of Ro.

2.1 Iterative Algorithms

The iterative algorithms build up the transitive closure in a breadth-first fashion. The number of iterations depends on the depth of the relation graph.

2.1.1 Semi-Naive Algorithm

In each iteration of Algorithm 2.1 (given below) the new tuples generated in the previous iteration are joined with all tuples of the original relation. In graph theoretic terms, in the ith iteration, all paths of length "i" are generated.

Algorithm 2.1

```
Input:      The source relation Ro
Output:     T holds the transitive closure of Ro after the algorithm terminates.
Method:
    begin
        T := Ro;
        $R\Delta$ := Ro;
        while $R\Delta \neq \emptyset$ do
        begin
            $R\Delta := R\Delta \bullet Ro$;
            $R\Delta := R\Delta - T$;
            $T := R\Delta \cup T$;
        end
    end.
```

After i iterations T will contain $\bigcup_{0 \le j \le i+1} R^j$.

Algorithm 2.1 terminates after p iterations, where p is the depth of relation graph Ro.

In each iteration, new tuples generated in the previous iteration are composed with the entire original relation. Also the entire result relation must be read in to remove duplicates.

2.1.2 Logarithmic Algorithm

The logarithmic method reduces the number of iterations by computing more of the transitive closure in each iteration.

<u>Algorithm 2.2</u>

```
Input:      The source relation Ro
Output:     T holds the transitive closure of Ro after the termination of the algorithm.
Method:
    begin
        T := Ro;
        $R\Delta$ := Ro;
        Rx := Ro;
        while Rx $\neq \emptyset$ do
            begin
                $R\Delta := R\Delta \bullet R\Delta$;
                $T\Delta := T \bullet R\Delta$;
                $Rx := R\Delta - T$;
                $T := R\Delta \cup T$;
                $T := T\Delta \cup T$;
            end
    end.
```

After i iterations T will contain $\bigcup_{0 \le j \le 2^i-1} R^j$.

Algorithm 2.2 terminates when $2^i - 1 \ge p$. Therefore the algorithm requires $\lceil \log_2(p+1) \rceil$ iterations. There are several disadvantages to the logarithmic algorithm. The current result relation (T) is scanned three times in each iteration. Furthermore, this result will contain many tuples that are not relevant to the computation, but cause unnecessary overhead. Also, unnecessary processing is done when p is not a power of two.

Valduriez *et. al.*[8] proposed an auxiliary data structure called "Join Indices" to improve the performance of the above algorithms. With Join Indices, each tuple in the original relation is given an unique surrogate. These Join Indices (tuple identifiers) are used to create a binary relation called a Join Index. The transitive closure of the Join Index is then taken, instead of the transitive closure of the original relation. Join Indices have been shown to improve the performance of the Semi-Naive and Logarithmic algorithms. Join Indices are costly to maintain as they must either be updated every time the relation is changed, or completely recomputed whenever the transitive closure is needed. Once the transitive closure is computed the new realation must be recomputed from the Join Indices.

2.2 Recursive Algorithms

In contrast to the iterative algorithms, recursive algorithms build up the transitive closure in a depth-first fashion. The number of iterations is independent of the depth of the graph.

2.2.1 Warren's Algorithm

In Warren's algorithm, the relation Ro is represented by an N by N boolean adjacency matrix X.

Algorithm 2.3

Input: The source relation Ro
Output: T holds the transitive closure of Ro after the termination of the algorithm.
Method:

```
begin
    Derive X(N,N) from Ro;

    for  i := 2 to N do
        for j := 1 to i - 1 do
            if X(i,j) then
                for k := 1 to N do
                    X(i,k) := X(i,k) $\vee$ X(j,k);


    for  i := 1 to N-1 do
        for j := 1+1 to N do
            if X(i,j) then
                for k := 1 to N do
                    X(i,k) := X(i,k) $\vee$ X(j,k)

    Form X(N,N) generate the result relation T;
end.
```

Warren's algorithm is a modification of Warshall's algorithm. Warren's algorithm improves the performance when the entire matrix cannot fit into memory. Warren's algorithm requires only two passes, each over half the matrix, irrespective of the depth of the relation graph. Warren's algorithm has been shown to perform better than the iterative algorithms for large memory sizes (i.e. the adjacency matrix fits into memory). Its performance degrades rapidly for small memory sizes.

3. Hash-Based Transitive Closure (HBTC) Approach

In the hash-based transitive closure (HBTC) approach taken by us, the given relation Ro(A,B) is partitioned twice, based on its A values, and also its B values. The transitive closure is then computed in two phases: "Join phase" and "Union Difference phase." In the "Join phase," the composition of corresponding partitions is computed, and in the "Union Difference phase" the results obtained from all partitions in the "Join phase" are merged. Throughout the processing, we maintain the initial partitions based on the A values, the updated partitions based on the B values (i.e. the partitions where the result from the previous iteration is stored), and the current result based on the B values. As the computation proceeds, none of the tuples of a certain partition may contribute in the subsequent iterations. We refer to such a partition as an *inactive partition* and the other partitions as *active partitions*.

Algorithm 3, below, is the Semi-Naive version of the HBTC. The variable *active-partition-set* holds the index numbers of the active partitions which are involved in the ith iteration, 1≤i≤p.

Algorithm 3

Input: The source relation Ro
Output: T holds the transitive closure of Ro after the termination of the algorithm.
Method:

```
          begin
                   Ro(A,B) Partitioned based on the A values into $Ra_{1..n}$;
                   Ro(A,B) Partitioned based on the B values into $Rb_{1..n}$;
                  $Rt_{1..n} := Rb_{1..n}$;
                  T := ø;
                  active-partition-set := {1..n};

                  while $Rb_{1..n} \neq ø$ do
{JOIN PHASE}
                           $Rb'_{1..n}$ := ø;
```

```
            for k $\in$ to active-partition-set do
                if $Ra_k$ and $Rb_k \neq ø$ then
                    $Rb'_{1..n} := Rb'_{1..n} \cup (Rb_k \bullet Ra_k)$;

            $Rb_{1..n} := Rb'_{1..n}$;

{UNION DIFFERENCE PHASE}
            for k $\in$ active-partition-set do
                if $Rb_k \neq ø$ then
                    begin
                        $Rb_k := Rb_k - Rt_k$;
                        $Rt_k := Rb_k \cup Rt_k$;
                    end
                else
                    active-partition-set := active-partition-set - {k};

        for k := 1 to n do
            $T := T \cup Rt_k$;
    end.
```

Here, the result of $Rb_k \bullet Ra_k$ is placed into its appropriate partitions (by unioning it with Rb'). This process is done by hashing on the B values of the tuples obtained in $Rb_k \bullet Ra_k$ and placing the tuples into the appropriate partitions of Rb'. The number of iterations is p, which is the same as for the Semi-Naive algorithm.

Algorithm 4, below, is the Logarithmic version of the HBTC.

Algorithm 4

Input: The source relation Ro
Output: T holds the transitive closure of Ro after the termination of the algorithm.
Method:

```
    begin
         Ro(A,B) Partitioned based on the A values into $R\Delta a_{1..n}$;
         Ro(A,B) Partitioned based on the B values into $R\Delta b_{1..n}$;
        $Rt_{1..n} := R\Delta b_{1..n}$;
        $T\Delta_{1..n} := R\Delta b_{1..n}$;
        $Rx_{1..n} := R\Delta b_{1..n}$;
        T := ø;
        active-partition-set := {1..n};
        while $Rx_{1..n} \neq ø$ do
{JOIN PHASE}
            $R\Delta'_{1..n} := ø$;
            for k $\in$ to active-partition-set do
                if $R\Delta a_k$ and $R\Delta b_k \neq ø$ then
                    $R\Delta'_{1..n} := R\Delta'_{1..n} \cup (R\Delta b_k \bullet R\Delta a_k)$;
            $R\Delta a_{1..n} := R\Delta'_{1..n}$;
            $R\Delta b_{1..n} := R\Delta'_{1..n}$;

            $T\Delta'_{1..n} := ø$;
            for k $\in$ to active-partition-set do
                if $T\Delta_k$ and $R\Delta a_k \neq ø$ then
                    $T\Delta'_{1..n} := T\Delta'_{1..n} \cup (T\Delta_k \bullet R\Delta a_k)$;
            $T\Delta_{1..n} := T\Delta'_{1..n}$;

{UNION DIFFERENCE PHASE}
            $Rx'_{1..n} := ø$;
            for k $\in$ to active-partition-set do
                if $R\Delta b_k$ and $Rt_k \neq ø$ then
                    $Rx'_{1..n} := R\Delta b_k - Rt_k$;
            $Rx_{1..n} := Rx'_{1..n}$;

            for k $\in$ active-partition-set do
                if $R\Delta b_k$ and $T\Delta_k \neq ø$ then
                        $Rt_k := Rt_k \cup R\Delta b_k \cup T\Delta_k$;
                else
                    active-partition-set := active-partition-set - {k};

        for k := 1 to n do
            $T := T \cup Rt_k$;
    end.
```

The original relation, as well as each of the intermediate relations from the Logarithmic algorithm has now been broken into n partitions. As with the Semi-Naive version, only the necessary partitions are brought into memory which reduces much of the extra procesing done by the Logarithmic algorithm. The number of iterations is $\lceil \log_2(p+1) \rceil$, which is the same as for the Logarithmic algorithm.

3.1 Memory Usage

Given M pages of memory, in the "Join phase" of the Semi Naive version (algorithm 3), n' pages are used for the partitions of Rb' (the hash table for the new tuples generated in an iteration), with one page devoted to each active partition. The corresponding partitions of Rb_k and Ra_k (for some $k \in$ active-partition-set) are loaded into the remaining memory (M - n'). Hybrid hash is used to perform the join within this remaining memory. Whenever a page corresponding to a partition of Rb' is filled up, that page is written to the disk.

In the "Union Difference phase" of the Semi Naive verison, corresponding partitions of Rb_k and Rt_k (for some $k \in$ active-partition-set) are brought into the memory so that duplicates can be removed from Rb_k and the new tuples added to Rt_k. The entire memory is available for this computation.

Similar work is done in the Logarithmic version (algorithm 4) except that two joins, a union, and a difference operation are done in each iteration due to the extra processing.

3.2 Partitioning

The partitioning plays an important part in the algorithm. The number of partitions will depend on the available memory. For smaller memory sizes, multiple nodes in the relation graph will be combined into a single partition. If a partition contains multiple nodes x and y, it is possible that node x will drop out before node y, but the tuples for both node x and node y would have to be brought into memory until node y drops out (at this time the partition drops out). When the number of partitions equals one, the algorithm reduces to simple Semi-Naive.

When multiple nodes are stored in a single partition, a unique identifier would have to be added to the attribute values in the buckets. This way only those nodes with the same identifier in the corresponding Ra and Rb partition would be joined.

The partitioning is done only in the initial phase of the algorithm and active-partition-set is initialized to contain all partitions. Each partition is assigned a distinct set of nodes. The partitions are not redefined subsequently. As the computation proceeds, partitions begin to drop out.

3.3 Why Do Partitions Drop Out?

If a page corresponding to a partition does not have tuples after an iteration then there is no reason to keep that partition in memory during the "Join phase." Let's assume attribute value x (node of the graph labelled x) corresponds to partition Rb_k (for some k $\in$ active-partition-set). Let's also assume that partition Rb_k contains values y and z after iteration i. This implies that the length of the path from y to x and z to x is i. In the subsequent iteration we use this length i path to compute length i+1 paths for node x. But if partition Rb_k is empty then there are no length i paths entering node x to participate in the next iteration. With no length i paths entering node x, there cannot be any length i+1 paths entering node x. Therefore there cannot be any more longer paths entering node x. Thus the partition in question will not be kept in memoryfor further iterations. Lu[6] discusses the above phenomena. It is possible that a longer version of an already generated path may try to go through an inactive node. This duplicate path is implicitly eliminated by the inactive node without necessitating an explicit duplicate removal step.

If a partition Ra_k is empty, then that partition will never be active in the "Join phase" of any iteration, but the corresponding Rb_k may be active in the "Union Difference phase." This is a node that has no outgoing edge, but may have a number of incoming edges, and thus is still active.

3.4 Example

Consider the relation Ro shown in Figure 2. Here, we assume one node per partition. The initial state and the states in subsequent iterations and phases of the Semi-Naive version of the method (algorithm 3) are shown below.

Ro(A, B)

1	4
2	1
2	3
3	6
4	3
4	5
4	6
5	1

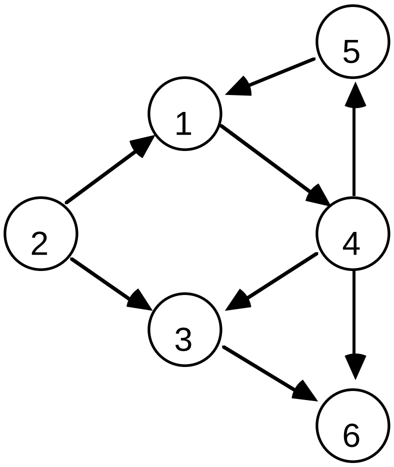

Figure 2

Relation Ro and its Corresponding Graph

Initialization State: (#active-partitions = 6)

Rb	Ra	Rt
2,5->1	1->4	2,5->1
ø->2	2->1,3	ø->2
2,4->3	3->6	2,4->3
1->4	4->3,5,6	1->4
4->5	5->1	4->5
3,4->6	6->ø	3,4->6

Ra$_1$ (1->4) shows that there is a path FROM node 1 TO node 4. R contains a tuple (1,4)
Rb$_1$ (2,5->1) shows that there is a path TO node 1 FROM nodes 2 and 5. R contains tuples (2,1) and (5,1)
Rt$_1$ (2,5->1) shows that there is a path TO node 1 FROM nodes 2 and 5. R contains tuples (2,1) and (5,1)

JOIN PHASE UNION DIFFERENCE PHASE

Iteration 1: (#active-partitions = 5)

Rb	Ra	Rb	Rt
2,5->1	1->4	4->1	2,4,5->1
ø->2*	*2->1,3	ø->2*	ø->2*
2,4->3	3->6	1->3	1,2,4->3
1->4	4->3,5,6	2,5->4	1,2,5->4
4->5	5->1	1->5	1,4->5
3,4->6*	*6->ø	1,2,4->6*	1,2,3,4->6*

Iteration 2: (#active-partitions = 5)

Rb	Ra	Rb	Rt
4->1	1->4	ø->1*	2,4,5->1*
ø->2*	*2->1,3	ø->2*	ø->2*
1->3	3->6	2,5->3	1,2,4,5->3*
2,5->4	4->3,5,6	ø->4*	1,2,5->4*
1->5	5->1	2->5	1,4->5
1,2->6*	*6->ø	1,2,5->6	1,2,3,4,5->6

Iteration 3: (#active-partitions = 3)

Rb	Ra
ø->1*	*1->4
ø->2*	*2->1,3
5->3	3->6
ø->4*	*4->3,5,6
2->5	5->1
5->6*	*6->ø

Rb	Rt
ø->1*	2,4,5->1*
ø->2*	ø->2*
ø->3*	1,2,4,5->3*
ø->4*	1,2,5->4*
ø->5*	1,4->5*
5->6	1,2,3,4,5->6

The computation terminates after iteration 3 because all partitions of Rb are empty.

In this example, inactive partitions are marked with an asterisk. Partition 6 never participates in the "Join phase" since Ra_6 is empty, but Rb_6 does get new tuples in each iteration. Partition 2 does not participate in anything since Rb_2 is empty. In the "Join phase" of iteration 3 a tuple 2->1 is generated, but partition Rb_1 is inactive during the "Union Difference phase" since Rb_1 partition is inactive during the "Join phase." Tuple 2->1 had been generated earlier. Since partition 1 is inactive, the duplicate (2,1) is implicitly removed without having to compare with t_1.

4. Cost Model

In this section we derive the cost formulae for the presented algorithms. These cost formulae are similar to those used by Lu[7] and Valduriez[8]. We assume that R(A,B) is a binary relation with the values of A, B coming from the same domain. The following set of parameters is used in deriving various cost measures:

M	= Number of pages of memory	
|R|	= Number of pages in relation R	
||R||	= Number of tuples in relation R	
JS	= Join Selectivity	$= \frac{\|\|R \text{ join } S\|\|}{\|\|R\|\| * \|\|S\|\|}$
US	= Union Selectivity	$= \frac{\|\|R \cup S\|\|}{\|\|R\|\| + \|\|S\|\|}$
DS	= Difference Selectivity	$= \frac{\|\|R - S\|\|}{\|\|R\|\|}$
F	= Hashing fudge factor	
TS	= Tuple length (in bytes)	
PS	= Page size (in bytes)	
p	= Number of iterations or the depth of the Transitive Closure	
HS	= Hash selectivity	
#part	= | active-partition-set |	
$\#part_o$	= initial | active-partition-set |	

t_{comp}	= Time for comparing two attribute values
t_{move}	= Time for moving a binary tuple in memory
t_{lookup}	= Time for looking up one value in the hash table
t_{hash}	= Time for hashing an attribute
t_{read}	= Time for reading one page from disk
t_{write}	= Time for writing one page to disk

HS is the fraction of partitions in iteration i which will be active in the iteration i + 1, for i= 1,2,...n. #part is the number of partitions that participated in iteration i. While ||R|| * ||S|| * JS gives the number of tuples in the next iteration, #part * HS gives us the number of partitions over which these tuples will be evenly distributed.

The appendix contains the cost for the Join(hybrid hash), Union, and Union Difference operations, as well as the cost of Semi-Naive, Logarithmic and Warren's algorithms. These are similar to those used by Lu[7] and Valduriez[8].

The join operation used by our algorithm is a slight modification of the hybrid-hash join method used by Lu[7] and Valduriez[8], we refer to this as HASHJOIN. In our algorithm a distinct page of memory is allocated for each of the partitions and the new tuples generated after each iteration are placed in their appropriate partitions. HASHJOIN does not deal with the cost required to write the new tuples generated at the end of each "Join phase" to the disk. This cost is related to the total number of new tuples generated in this iteration, not just the number of new tuples generated for this partition. So, while this cost is incurred in the "Join phase," it is counted when we write the new tuples to the disk. We refer to this distribution as HASHDISTRIBUTION.

```
HASHJOIN(R,S)
STEP  1-2 of JOIN(R,S)
STEP  3        if $(||R|| + ||S||) * F * TS > (M-\#part)$ then
                   begin
                   $+ ((||R|| + ||S||) * TS - (M - \#part) /F) / TS * t_{move}$
                   $+ ((||R|| + ||S||) * TS - (M - \#part) /F) / PS * t_{write}$
                   $+ ((||R|| + ||S||) * TS - (M - \#part) /F) / PS * t_{read}$
                   $+ ((||R|| + ||S||) * TS - (M - \#part) /F) / TS * t_{hash}$
                   end;
STEP  4-5 of JOIN(R,S)
```

As new tuples are generated in the "Join phase" they will be hashed and moved into the appropriate partition. As the number of new tuples is based on the total number of tuples in the partitions being joined this calculation must be done separate from the "Join phase" in what is called the "Hashdistribution phase."

```
HASHDISTRIBUTION
STEP  1        $2 * ||R|| * ||S|| * JS * t_{hash}$;
STEP  2        $+ 2 * ||R|| * ||S|| * JS * t_{move}$;
```

The cost of moving these tuples from the memory to the disk is given below.

$$\text{STEP 3} \quad + \#part * HS^{i} * \left\lceil \frac{||R|| * ||S|| * JS}{\#part * HS^{i}} * \frac{TS}{PS} \right\rceil * t_{write}$$

4.1 Cost of HBTC Algorithm

We assume that the new tuples generated in iteration i are evenly distributed across the active partitions for iteration i+1 for i = 0..p, and that the number of active partitions decreases by a factor of HS after each iteration.

The number of partitions must be chosen upon entry into the algorithm. There must be enough room in memory to store the hash tables for Rb_k and Ra_k, as well as one page for each partition to hold new tuples. Given the available memory as M pages, the initial number of partitions is determined by the following equation:

$$M = \#part_o + \frac{2 * |Ro| * F}{\#part_o}$$

Solving for $\#part_O$ in the above we get:

$$\#part_o = \frac{M + \sqrt{M^2 - 8 * |Ro| * F}}{2}$$

The maximum number of partitions is obtained when each partition contains exactly one distinct node; each node representing a unique value in the relation Ro. Therefore $\#part_O$ can not be any greater than #nodes. Since the directed graph(relation) could vary from a complete directed graph to a chain, #nodes can vary from $\sqrt{2 * \#edges + 1}$ to #edges + 1, respectively. In our cost analysis, where ||Ro|| = #edges, #nodes has been set to:

$\#nodes = 2 * \sqrt{2 * \|Ro\|}$ which is twice the number of nodes in the complete graph.

This number of active partitions is to ensure that the partitions in the first iteration fit in memory. $|Ra_k|$ remains constant, but $|Rb_k|$ fluctuates as computation proceeds. Also, #part decreases. Since only the active partitions are needed in the "Join phase," there will be additional room in memory to store the hash tables for Rb_k and Ra_k;if partitions become inactive.

The cost (C) of the HBTC approach to the Semi-Naive algorithm is given below:

$$\begin{aligned}
C :=& \\
&|Ro| * t_{read} \\
&+ \|Ro\| * 2 * t_{hash} \\
&+ |Ro| * 3 * t_{write} \\
&+ \sum_{i=1}^{p} \left[\begin{array}{l} \displaystyle\sum_{k=1}^{\lceil \#part_o * HS^{i-1} \rceil} \mathrm{HASHJOIN}\left(Ra_k^{i-1}, Rb_k^{i-1}\right) \\ + \mathrm{HASHDISTRIBUTION} \\ + \displaystyle\sum_{k=1}^{\lceil \#part_o * HS^{i} \rceil} \mathrm{Union\_Difference}\left(Rb_k^{i}, t_k\right) \end{array} \right] \\
&+ \sum_{i=1}^{\#partitions} Rt_i * \left(t_{read} + t_{write}\right)
\end{aligned}$$

The cost (C) of the HBTC approach to the Logarithmic algorithm is given below:

$$
\begin{aligned}
&C := \\
&|Ro| * t_{read} \\
&+ ||Ro|| * 2 * t_{hash} \\
&+ |Ro| * 5 * t_{write} \\
&+ \sum_{i=1}^{\lceil \log_2(p+1) \rceil} \left[\begin{aligned}
& \sum_{k=1}^{\lceil \#part_0 * HS2^{i-1} \rceil} \mathrm{Join}\left(R\Delta_{bk}, R\Delta_{ak}\right) \\
& + 2 * \mathrm{HASHDISTRIBUTION} \\
& \sum_{k=1}^{\lceil \#part_0 * HS2^{i-1} \rceil} \mathrm{Join}\left(T\Delta_{k}, R\Delta_{ak}\right) \\
& + \mathrm{HASHDISTRIBUTION} \\
& + \sum_{k=1}^{\lceil \#part_0 * HS2^{i} \rceil} \mathrm{Union\_Difference}\left(R\Delta_{bk}, Rt_{k}\right) \\
& + \sum_{k=1}^{\lceil \#part_0 * HS2^{i} \rceil} \mathrm{Union}\left(T\Delta_{k}, Rt_{k}\right)
\end{aligned} \right] \\
&+ \sum_{i=1}^{\#partitions} |Rt| * \left(t_{read} + t_{write}\right)
\end{aligned}
$$

5. Performance Comparison

In this section we compare the performance of HBTC Semi Naive, Semi-Naive, HBTC Logarithmic, Logarithmic, and Warren's algorithms. The following set of values are assumed throughout the comparisons.

M	-	7,500
|Ro|	-	10,000
|R|	-	30,000
JS	-	10^{-7}
US	-	1.0
DS	-	1.0
F	-	1.2
TS	-	8 Bytes
PS	-	1600 Bytes
P	-	24
HS	-	0 .9
t_{comp}	-	3 microsecs
t_{move}	-	20 microsecs
t_{hash}	-	9 microsecs
t_{lookup}	-	6 microsecs
t_{read}	-	15 millisecs
t_{write}	-	20 millisecs

For the comparison, we have chosen to vary three parameters: Depth, Memory Size, and HS.

Figure 3 illustrates HS versus execution time in the HBTC Semi-Naive and HBTC Logarithmic algorithms. Values for the other three algorithms are shown for comparison. When HS is one, none of the partitions drop out. HBTC Semi-Naive still performs better than Semi-Naive, and HBTC Logarithmic still performs better than Logarithmic because partitioning means fewer tuples must be written back to the disc during the "Join phase." When HS decreases below 0.95, HBTC Semi-Naive begins performing better than the other three non-HBTC algorithms because of partition dropout. HBTC Logarithmic does not have such a drastic change in behavior since it has logarithmically fewer iterations. Below 0.5 the performance is relatively constant.

Figure 4 illustrates number of partitions versus execution time in the HBTC Semi-Naive and HBTC Logarithmic algorithms. The number of partitions is usually set according to the formulas in Section 4.4, but here the number of partitions is varied from one to the maximum number of partitions for a relation of that size. When the number of partitions is equal to one, the HBTC algorithms perform like the non HBTC versions. The partitioning of the source relation creates dramatic improvements with only a few partitions. The execution time continues to improve as the number of partitions are increased until the number of partitions reaches approximately 1000. At this point there is a slight performance decrease since not enough partitions drop out to give enough room to bring all of the partitions to be joined into memory at once within the first few iterations.

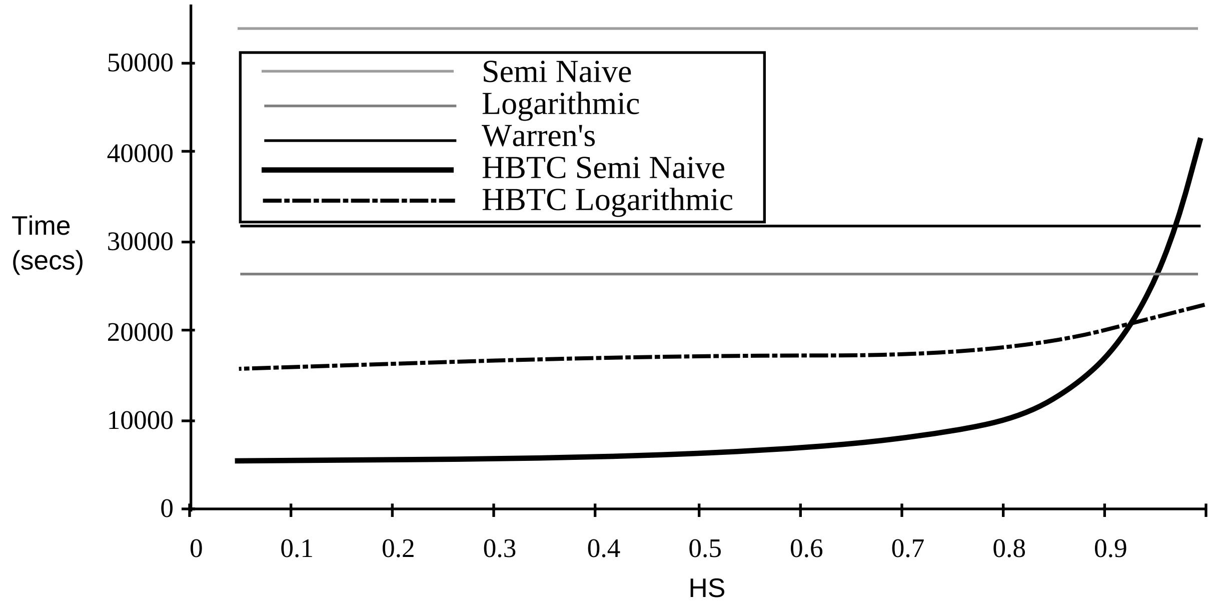

Figure 3
HS versus Execution Time

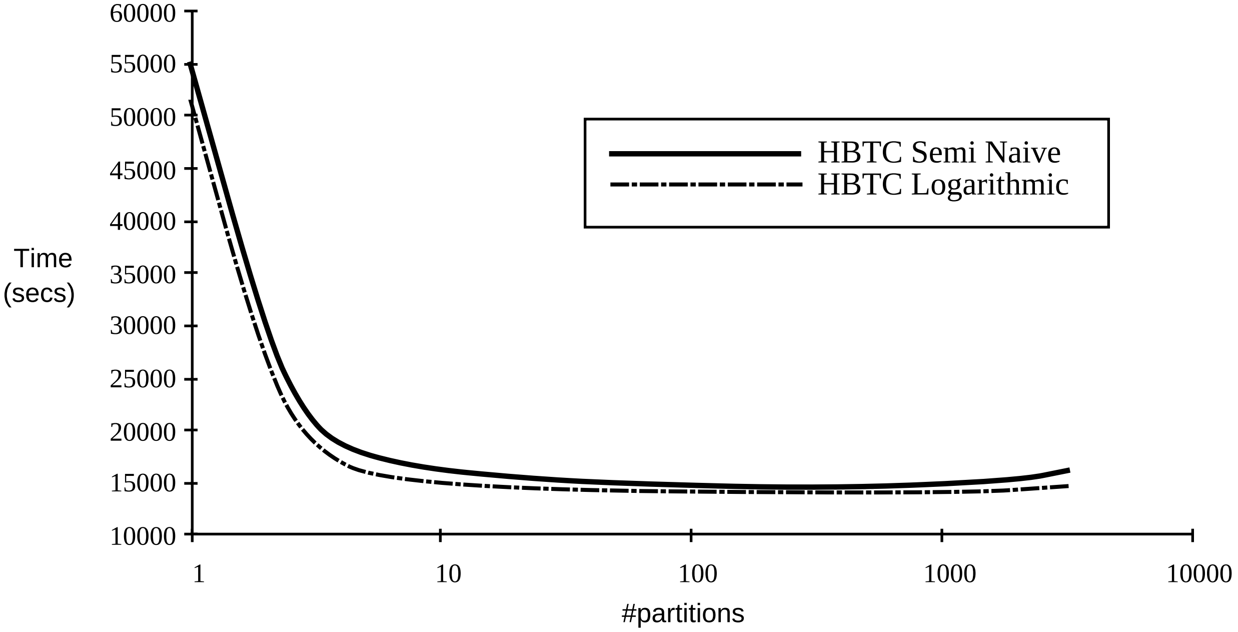

Figure 4
#partitions versus Execution Time

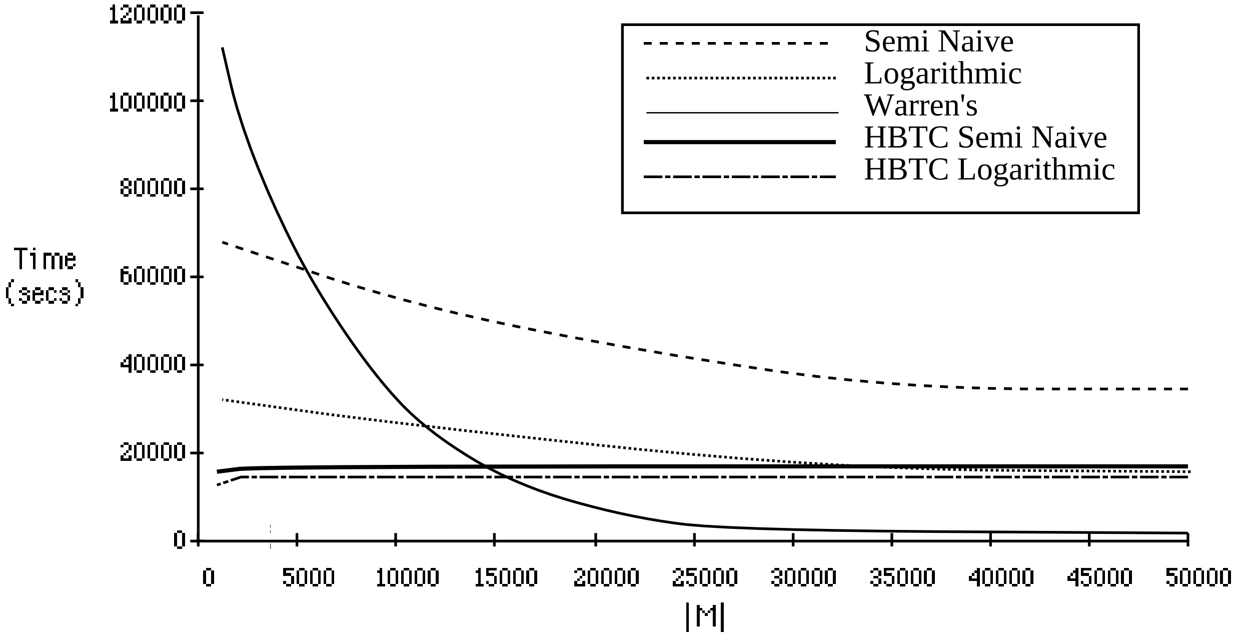

Figure 5
Memory size versus Execution Time

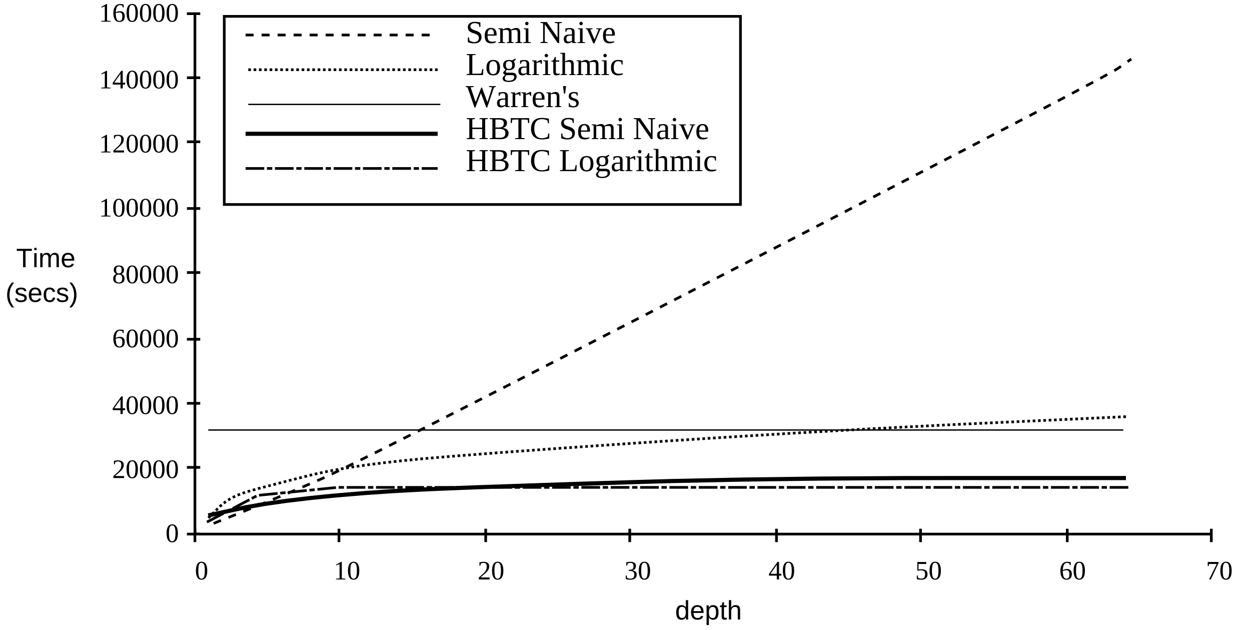

Figure 6
Depth versus Execution Time

Figure 5 illustrates memory size versus execution time for all five algorithms. Since HBTC Semi-Naive and HBTC Logarithmic deals with many small partitions instead of a single large partition, their performance is almost constant over a very wide range of memory sizes. When memory size is smaller than or comparable to the relation size, the HBTC algorithms do significantly better than the other three methods. Only when memory size is more than double the size of the relation size does Warren's algorithm perform better than the HBTC algorithms. When the memory size is four times as large as the relation size, the logarithmic algorithm begins to perform slightly better than HBTC algorithms.

Figure 6 illustrates depth versus execution time for all five algorithms. Warren's execution time remains constant as the depth increases, but since the memory size is near the relation size, it does not perform as well as the HBTC algorithms. HBTC's performance remains almost constant with changing depth due to the partition drop out.

6. Conclusions

We have presented and analyzed a hash-based method to compute the transitive closure of a database relation. The algorithm uses hashing to partition the source relation. Hashing is also needed to place the new tuples generated after each iteration of the algorithm into appropriate partitions. We have shown that this method can be applied to both the Semi Naive and Logarithmic algorithms to improve performance over a wide range of parameter values. Since the partitions are disjoint, much of the processing could be done concurrently, and thus our method is amenable to parallelization. Currently we are investigating the use of this technique in computing aggregative closure and traversal recursions.

References

1. R. Agrawal and H. Jagadish, Direct algorithms for computing the transitive closure of database relations. *Proc. 13th VLDB Conf., Sept. 1987.*

2. D DeWitt, R. Katz, F. Oken, L. Shapiro, M. Stonebraker and D. Wood, Implementation techniques for main memory database systems. *Proc. ACM-SIGMOD Conf., June 1984.*

3. Y.E. Ioannidis, On the computation of the transitive closure of relational operations. *Proc. 12th VLDB Conf, Aug. 1986.*

4. H.V. Jagadish, R. Agrawal, and L. Ness, A study of transitive closure as a recursion mechanism. *Proc. ACM-SIGMOD Conf., May 1987.*

5. A. Johnson, Recursive query processing in deductive databases. *MS thesis*, Computer Science Dept, Wayne State University, Detroit, Micigan, USA, 1989, (in preparation).

6. H. Lu, New strategies for computing the transitive closure of a database relation. *Proc. 13th VLDB Conf., Sept. 1987.*

7. H. Lu, K. Mikkilineni, and J.P. Richardson, Design and evaluation of algorithms to compute the transitive closure of a database relation. *Proc. Third Intl. Conf. on Data Engineering, Feb. 1987.*

8. P. Valduriez and H Boral, Evaluation of recursive queries using join indices. *Proc. First Intl. Conf. on Expert Database Systems, Apr. 1986.*

Appendix:

Cost of Basic Operations

JOIN(R,S)

```
STEP 1    $(|R| + |S|) * t_{read}$
STEP 2    $+ (||R|| + ||S||) * t_{hash}$
STEP 3    if $(||R|| + ||S||) * F * TS > m$ then
              begin
              $+ ((||R|| + ||S||) * TS - M * PS / F) / TS * t_{move}$
              $+ ((||R|| + ||S||) * TS - M * PS / F) / PS * t_{write}$
              $+ ((||R|| + ||S||) * TS - M * PS / F) / PS * t_{read}$
              $+ ((||R|| + ||S||) * TS - M * PS / F) / TS * t_{hash}$
              end
STEP 4    $+ ||R|| * t_{move}$
STEP 5    $+ ||S|| * t_{comp}$
STEP 6    $+ 2 * ||R|| * ||S|| * JS * t_{move}$
STEP 7    $+ ||R|| * ||S|| * JS * TS / PS * t_{write}$
```

UNION(R,S)

```
STEP 1-5 of JOIN(R,S)
STEP 6    $+ (||R|| + ||S||) * US * t_{move}$
STEP 7    $+ (||R|| + ||S||) * US * TS / PS * t_{write}$
```

UNION DIFFERENCE(R,S)

```
STEP 1-7 of UNION(R,S)
STEP 8    $+ (||R|| + ||S||) * DS * t_{move}$
STEP 9    $+ (||R|| + ||S||) * DS * TS / PS * t_{write}$
```

Cost (C) of Semi-Naive Algorithm

$$C :=$$

$$|Ro| * t_{read}$$

$$+ |Ro| * 2 * t_{write}$$

$$+ \sum_{i=1}^{p} \left[\begin{array}{l} \mathrm{Join}\left(R\Delta_i,\ R_0\right) \\ + \mathrm{Union\_Difference}\left(R\Delta_i, T\right) \end{array} \right]$$

Cost (C) of Logarithmic Algorithm

$$C :=$$

$$|Ro| * t_{read}$$

$$+ |Ro| * 2 * t_{write}$$

$$+ \sum_{i=1}^{\lceil \log_2(p+1) \rceil} \left[\begin{array}{l} \mathrm{Join}\left(R\Delta_i,\ R\Delta_i\right) \\ + \mathrm{Join}\left(T,\ R\Delta_i\right) \\ + \mathrm{Union\_Difference}\left(R\Delta_i,\ T\right) \\ + \mathrm{Union}(T_i,\ T) \end{array} \right]$$

Cost (C) of Warren's Algorithm

$$
\begin{aligned}
C :=\ & \\
& |R_0| * t_{read} \\
& + \|R_0\| * (t_{hash} + t_{move}) \\
& + |R_0| * \left(1 - \frac{M}{|R_1|}\right) * t_{write} \\
& + \ |R_0| \ * \frac{PS}{TS} * \log_2\left(\frac{PS}{TS}\right) \ * \ (t_{comp} + t_{move})
\end{aligned}
$$

Pass #1

$$
\begin{aligned}
& + |R_0| * \left(1 - \frac{M}{|R_1|}\right) * t_{read} \\
& + \|R_1\| \ * t_{comp} \\
& + (\|R_1\| - \|R_0\|) * \ (t_{hash} + t_{lookup}) \\
& + \sum_{i = \frac{M * PS}{TS} + 1}^{\|R_1\|} \frac{i * TS - M * PS}{i * TS} * t_{read} \\
& + (\|R_1\| - \|R_0\|) * \log_2\left(\frac{PS}{TS}\right) * \ t_{comp} \\
& + (\|R_1\| - \|R_0\|) * \ t_{move} \\
& + \left(\|R_1\| - \frac{M * PS}{TS}\right) * \ t_{move} \\
& + (|R_1| - M) * \ t_{write}
\end{aligned}
$$

Pass #2

$$
\begin{aligned}
& + \ (|R_1| - M) * t_{read} \\
& + \|R\| \ * t_{comp} \\
& + (\|R\| - \|R_1\|) * \ (t_{hash} + t_{lookup}) \\
& + \sum_{i = \frac{M * PS}{TS} + 1}^{\|R\|} \frac{i * TS - M * PS}{i * TS} * t_{read} \\
& + (\|R\| - \|R_1\|) * \log_2\left(\frac{PS}{TS}\right) * \ t_{comp} \\
& + (\|R\| - \|R_1\|) * \ t_{move} \\
& + \ \|R\| * \ t_{move} \\
& + |R| * \ t_{write}
\end{aligned}
$$