

# DIRECT ALGORITHMS FOR COMPUTING THE TRANSITIVE CLOSURE OF DATABASE RELATIONS

*Rakesh Agrawal  
H. V. Jagadish*

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

We present new algorithms for computing the transitive closure of large database relations. Unlike iterative algorithms, such as the semi-naive and the logarithmic algorithms, the termination of our algorithms does not depend on the length of paths in the underlying graph (hence, the name *direct* algorithms). We also present simulation results that show that these direct algorithms perform uniformly better than the best of the iterative algorithms. A side benefit of this work is that we have proposed a new methodology for evaluating the performance of recursive queries.

## 1. INTRODUCTION

With the increasing "non-traditional" uses of relational databases, several extensions have been proposed to the relational query languages in order to efficiently support these applications. A common operator that appears in many of these proposals is the transitive closure operation (see, for example, Zloof's QBE [17], Guttman's \* extension to Quel [7], Probe's traversal recursion [11], and Agrawal's  $\alpha$ -extended relational algebra [1]). In [9], it has been shown that every linearly recursive query can be expressed as a transitive closure possibly preceded and followed by operations already available in relational algebra, once again emphasizing the importance of

transitive closure as a primitive database operation.

Much of the success of the relational database systems can be attributed to the discovery of efficient algorithms for implementing various relational operators. A similar research effort is required into investigation of algorithms for computing transitive closure of large database relations. We categorize the known transitive closure algorithms into *iterative* algorithms and *direct* algorithms. Examples of iterative algorithms include the semi-naive [3] and the logarithmic [8,14] algorithms, developed in the context of evaluation of general recursive queries. These algorithms do not utilize the special structure of a transitive closure problem. Direct algorithms, on the other hand, do not view the problem as one of evaluating a recursion, but rather obtain the closure from first principles. These algorithms were originally presented in a different context and expect the starting point to be a Boolean matrix. Examples of direct algorithms include, among others, Warshall's algorithm [16], Warren's algorithm [15], Schmitz's Algorithm [12], and Schnorr's algorithm [13].

In this paper, we present new algorithms for implementing transitive closure in the database context. These algorithms were obtained by modifying the existing direct techniques for computing transitive closure, and by combining more than one technique in some cases. We also evaluate the performance of these algorithms against the iterative algorithms, and show that, for a large range of underlying datasets, these direct algorithms perform better than the best of the iterative algorithms. In many cases we were able to show an improvement several orders of magnitude.

The organization of the rest of the paper is as follows. In Section 2, we briefly describe some well-known direct algorithms for computing transitive closures of a Boolean matrix, and also show how these algorithms could be used in the context of relational databases. Section 3 is the heart of this paper where we present

---

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

new direct algorithms for computing the transitive closure of large database relations. In Section 4, we propose a methodology for evaluating the performance of recursive queries, and evaluate the performance of the algorithms developed in Section 3 against the best iterative algorithm. Finally, in Section 5, we summarize the main conclusions of this study.

## 2. NAIVE DIRECT ALGORITHMS

In this section, we give a brief overview of some of the popular algorithms that were originally proposed to compute the transitive closure of Boolean matrices. The new algorithms that we propose in Section 3 have been inspired by these algorithms. We also indicate for the purposes of comparison how one could obtain a straightforward adaptation of these algorithms to compute the transitive closure of database relations.

### 2.1 The Warshall Algorithm

Given an initial  $v \times v$  Boolean matrix of elements  $a_{ij}$  over a  $v$  node graph, with  $a_{ij}$  being 1 if there is an arc from node  $i$  to node  $j$  and 0 otherwise, its transitive closure can be obtained as [16]:

```

For  $k=1$  to  $v$ 
  For  $i=1$  to  $v$ 
    For  $j=1$  to  $v$ 
       $a_{ij} = a_{ij} \vee (a_{ik} \wedge a_{kj})$ 

```

If a graph is represented as a relation with each tuple representing an arc, the Warshall algorithm can be implemented in the following manner. For each node  $n$ , first fetch its successor list. Then for each predecessor  $p$  of  $n$ , fetch the successor list of  $p$ , and add to the successor list of  $p$  the successor list of  $n$  (removing duplicates if any). In order to determine the predecessors of  $n$ , the successor list of all other nodes may be scanned to see if  $n$  appears in them. An alternative would be to maintain, in addition to the successor list, a predecessor list also with each node. In that case, the determination of the predecessors of  $n$  would become trivial, but at the time of updating the successor list of the predecessor  $p$ , the predecessor list associated with each of the successors of  $n$  must also be updated to include  $p$  in them.

### 2.2 The Warren Algorithm

Warren [15] noted that the Warshall algorithm involves fetching random bits from the Boolean matrix, and proposed a modification that would permit direct operation upon Boolean vectors without the overhead of bit extraction:

```

For  $i=1$  to  $v$ 
  For  $k=1$  to  $i-1$ 
    For  $j=1$  to  $v$ 
       $a_{ij} = a_{ij} \vee (a_{ik} \wedge a_{kj})$ 
  For  $i=1$  to  $v$ 
    For  $k=i+1$  to  $v$ 
      For  $j=1$  to  $v$ 
         $a_{ij} = a_{ij} \vee (a_{ik} \wedge a_{kj})$ 

```

The only change is that the  $i$  and  $k$  loops have been interchanged. However, this interchange could result in some paths being missed out and so the algorithm now requires two "passes" before it completes. The modification in the range of the second loop index,  $k$ , is an optimization that reduces the cost of two passes.

A straightforward database implementation of the Warren algorithm can readily be derived. Sort the existing database to form a successor list. Then, for every node, fetch its successor list, and for every successor of this node, fetch the successor list in turn and add it to the successor list of the original node, avoiding duplicates.

### 2.3 Other Direct Algorithms

Schnorr's algorithm [13] and Schmitz's Algorithm [12] are two other popular direct algorithms for the computation of transitive closure. However, a drawback of these techniques is that they may often simply state that there exists a path between two nodes without actually stating what the path is. If one is also interested in a relationship between the two nodes that is path dependent (such as shortest path, for example), these algorithms cannot be used. In view of this difficulty, we decided not to consider Schnorr, Schmitz, and similar algorithms.

## 3. EFFICIENT DIRECT ALGORITHMS

In this section, we present some modifications of the algorithms discussed in the previous section, and suggest some non-obvious implementations that we expect will perform much better than the naive implementations discussed above. All the algorithms below assume that the initial relation has been sorted on the fields participating in the transitive closure, so that all the "successors" of a given node can be found in a contiguous set of tuples in the relation.

We assume that the relation whose transitive closure is to be computed is large compared to the memory available, and must be partitioned into pieces each of which can fit in memory. A partition will consist of the successor lists of several nodes. These lists grow as the closure is computed, and the initial partitions may no longer fit in the memory. To handle this

situation, all of the algorithms below rely on *dynamic partitions* — there is no need to create partitions ahead of time. When it is time to read a partition into memory, enough tuples are read to fill a pre-defined fraction of memory. As the computation proceeds, if the memory starts filling up, some successor lists are deleted or written back out of memory to be included with the next partition that is read in.

We find it convenient to continue to think of the transitive closure as a matrix problem. To facilitate understanding of the algorithms, they will first be presented as if the partitions were pre-defined, static, and never adjusted. The re-partitioning discussed in the previous paragraph will be dealt with towards the end of the discussion of each algorithm.

Our objective in each of the algorithms presented below is to recognize the constraint that the entire relation cannot reside in the main memory all at once. Given that only one partition (or a *block*, as we find it convenient to call it) resides in the memory, we wish to devise direct algorithms that minimize the input-output required for tuples that are not in the memory. Our approach will be to begin with the Warshall algorithm described in Section 3, and to reorder the computations performed satisfying the two precedence constraints (see [2] for the derivation of these constraints):

1.  $(i,k)$  precedes  $(i,j)$  for all  $k < j$ , for all  $i$
2.  $(j,k)$  precedes  $(i,j)$  for all  $k < j$ , for all  $i$

Where by  $(i,j)$  we mean “the examination of whether there is an arc from node  $i$  to node  $j$  (that is, the tuple  $\langle i,j \rangle$  exists) and the possible action consequent upon finding that there is one (adding the successor list of  $j$  to the successor list of  $i$ , with duplicates eliminated)”. We shall refer to such examination and possible update as the *processing* of element  $(i,j)$ .

### 3.1 The Blocked Warshall Algorithm

We motivate the rationale of the Blocked Warshall algorithm through an example. Suppose that the graph is such that there are arcs from node 3 to nodes 1 and 2. Normally, the Warshall algorithm processes all the matrix elements column by column, that is, it will first process node 1, then node 2, and so on. For processing node 1, all the successors of 1 are first fetched into the memory, and then all the predecessors of 1 are examined one after the other. So, the successors of 3 will be fetched and to this successor list, the successors of 1 will be added. Let

us assume that 1 has many predecessors, so at some stage of processing of 1, the successor list of 3 is paged out of the memory. After the processing for node 1 has finished, the successors of 2 will be fetched, and then the predecessors of 2 will be examined. Consequently, the successor list of 3 will have to be again paged in from disk into memory. To minimize this I/O traffic, we would like, if possible, to consider the nodes 1 and 2 together, so that when the successor list of 3 is brought into memory, successors of both 1 and 2 are added to it. In other words, instead of processing one column at a time, we would like to process a block of columns at a time. A block of column constitutes a partition, and within a partition, we would like to process row-wise, that is, after examining  $(3,1)$ , we would like to examine  $(3,2)$  before going over to  $(4,1)$ .

Attempting to examine elements in this order, we find that after processing element  $(1,1)$ , we cannot even examine  $(1,2)$  without violating the second precedence constraint ( $(2,1)$  should be examined before  $(1,2)$ ). Let us take a step back and review the situation to come up with a possible remedy. When element  $(i,j)$  is processed, we guarantee that the successor list of  $j$  is in memory. If  $i$  does not belong to the same partition as  $j$ , all the arguments of the previous paragraphs apply, and we would like to proceed row-wise for the row  $i$  to minimize fetches to such  $i$  lists. However, if  $i$  is in the same partition as  $j$ , then the successor lists of both  $i$  and  $j$  are already in memory, and neither has to be fetched when element  $(i,j)$  is examined. Therefore, for all such  $i$  in the same partition as  $j$ , the I/O is not affected by the order in which the nodes are processed. In particular, we could process such “diagonal block” elements column-wise rather than row-wise, without degrading performance. (The “diagonal block” consists of elements whose row numbers and whose column numbers both lie within the range of the partition currently in memory).

Another point to notice is that if  $(i,j)$  is in the diagonal block, then all  $(j,k)$  that are material to the second precedence constraint are in the same set of rows as the diagonal block and in a column to the left of  $j$ . The first precedence constraint always applies only between elements on the same row. Therefore, given a column partition that is to be processed and that all the column partitions to the left of this partition have been processed, we may first process the diagonal block (doing so column-wise as discussed above), without violating any precedence constraints.

Having processed the diagonal block in a column partition, we claim that the elements in the off-diagonal rows<sup>1</sup> within this column partition may be processed row-wise. Row-wise processing automatically satisfies the first precedence constraint. All  $(j,k)$  that are now relevant to the second constraint are in a row that belongs to the diagonal block and in a column that is in the diagonal block or to the left of it, and the second constraint is also satisfied. Another consequence of this observation is that it is not necessary to process the off-diagonal rows in any particular order (like top to bottom) — they may be processed in any convenient order.

One can now write the Blocked Warshall algorithm:

**Algorithm 1 : Blocked Warshall**

```

For each column partition
    (columns  $j_b$  to  $j_e$  inclusive)

    /* Processing of diagonal block */
    For  $j = j_b$  to  $j_e$ 
        For  $i = j_b$  to  $j_e$ 
            If tuple  $\langle i,j \rangle$  exists
                Add succ. list  $j$  to succ. list  $i$ 

    /* Processing of off-diagonal rows */
    For  $i = 1$  to  $v \wedge i \notin j_b$  to  $j_e$ 
        For  $j = j_b$  to  $j_e$ 
            If tuple  $\langle i,j \rangle$  exists
                Add succ. list  $j$  to succ. list  $i$ 

```

Figure 1 shows a  $7 \times 7$  matrix and the order in which the elements of this matrix will be processed using the Blocked Warshall algorithm. The vertical lines bracket the column partitions, and the horizontal lines together with the vertical lines delineates the diagonal blocks. Notice that the order of computation is significantly different from the straight Warshall algorithm.

1	3	24	25	26	40	41
2	4	27	28	29	42	43
5	6	15	18	21	44	45
7	8	16	19	22	46	47
9	10	17	20	23	48	49
11	12	30	31	32	36	38
13	14	33	34	35	37	39

**Figure 1.** The order of computation in the Blocked Warshall algorithm

**Dynamic Partitioning**

Notice that Blocked Warshall first reads a set of successor lists corresponding to a column partition into the memory, and processes the diagonal block. After the diagonal block processing, one off-diagonal row at a time is read and processed. During the processing of a off-diagonal row, the successor lists corresponding to the rows in the diagonal blocks may be added to the successor list corresponding to the current off-diagonal row. However, before the next off-diagonal row is read, the current off-diagonal row is written out to the disk. Therefore, except during the processing of the diagonal block, there is little addition to the contents of memory. However during the processing of the diagonal block, it is possible that the successor lists in memory grow until the memory becomes full. If the partitioning were static, such an unanticipated growth could be catastrophic and could result in forcing the entire algorithm to be re-executed with smaller partitions. We would like to be able to dynamically alter the partitions when the need arises.

During the diagonal block processing, the size of the partition may be dynamically reduced by discarding the successor list corresponding to the last column and including it in the next partition instead. If further memory space is needed, the same discard procedure could be repeated whenever and as often as required. The only constraint is that at the time of the discard, one should not yet have begun processing the column being discarded (or if such processing has commenced, one should be able to undo its effects). We thereby guarantee that none of the successor lists not discarded could have been affected as a result of the discarded node. (During the column-wise processing of a diagonal block, the successor list of the node corresponding to a column is not added to any other successor list, unless the processing of that column begins). However, in processing the diagonal block column-wise, some elements in the discarded row (corresponding to the column being discarded) could have been processed, updating the discarded successor list. This update is immaterial, since the

1. We will use the phrase "off-diagonal  $\langle$ entity $\rangle$ " to refer to an  $\langle$ entity $\rangle$  not in the diagonal block.

rows not in the diagonal block will shortly be processed any way. We have two choices. We can undo the effect of the updates by simply not writing back the discarded successor list, and recompute these updates when this list is later read back for off-diagonal row processing. Alternatively, we could write back the discarded successor list, remember till what point it has already been updated, and save on some computation when this successor list is read back for row-wise processing.

### 3.4 The Blocked Warshall Algorithm with Predecessor Lists

One drawback of the Blocked Warshall algorithm (or indeed of even the plain Warshall if implemented only with successor lists) is that processing an off-diagonal element  $(i, j)$  involves determining whether  $j$  is a successor of  $i$ , and to make this determination one has to look at the successor list of  $i$  which is not in memory. Therefore, the successor list of every off-diagonal row  $i$  is always fetched into the memory whether or not the successor list is updated. The successor list of some diagonal-block row  $j$  (which is already in memory) is added to  $i$  only if indeed  $j$  is a successor of  $i$ . If the predecessor list of  $j$  was also available in memory along with its successor list, only those  $i$  need be fetched that actually have to be updated and one can eliminate some wasteful I/O. This also implies that during the processing of a column partition, the predecessor lists of only those nodes whose successor lists are in the current diagonal block need to be present in the memory.

One has to perform some initial effort in forming the predecessor lists from the original relation. The storage requirement for the predecessor list is not necessarily severe. Only the two fields involved in the transitive closure need to participate in the predecessor list. If each tuple in the relation consists of several extraneous fields not directly participating in the determination of the transitive closure, it is possible that the size of the predecessor lists is only a small fraction of the size of the successor lists.

Besides the cost of I/O of the predecessor lists, the biggest extra cost of the Blocked Warshall with Predecessors algorithm is that every time one examines element  $(i, j)$  and adds the successor list of  $j$  to the successor list of  $i$ , one also has to add  $i$  to the predecessor list of every successor of  $j$ . Normally, predecessor lists of the successors of  $j$  would not be in memory (except for those successors which are in the diagonal block), and will have to be fetched from disk, updated, and written back to disk, a fairly expensive proposition. We would like to somehow reorganize the order of these updates to render them

more efficient. In particular, if possible, we would like to postpone all the predecessor updates to the end of the processing of a column partition, and then do all the updates for a node in one shot. We had noted earlier that the predecessor lists of all the nodes in the diagonal block has to be kept in the memory during the whole processing of a column partition. We can, therefore, allow continuous update of such lists without degradation in performance.

The algorithm given below realizes the above objective. One may be led into thinking that Blocked Warshall with Predecessors would be a straightforward adaptation of Blocked Warshall, but the predecessor-update optimization makes it an interesting and rather unintuitive algorithm. For a column partition, as in Blocked Warshall, this algorithm first processes the diagonal block, and then the off-diagonal rows. In the off-diagonal processing, we split the update of successor lists and the update of predecessor lists, performing each in its entirety. Let us call these three steps Phases I, II, and III respectively. During Phases I and II, the predecessor lists (already in memory) of the nodes in the diagonal block are always kept current. However, the predecessor list (not in memory) of any off-diagonal node is not updated. We can defer this update because during the processing of a column partition, the predecessor list of any off-diagonal node is not consulted. In Phase III, the predecessor lists of the off-diagonal nodes are updated. What is interesting is that for this update only the rows (successor lists) in the current diagonal block (which are already in the memory) need to be consulted. If an off-diagonal node is present in any of the successor lists, the predecessor list of this node is fetched from the disk and to it are added the predecessor lists (already in memory) of those nodes in whose successor list this node was found<sup>2</sup>. The reason for the correctness of this procedure is that for any new arc  $(i, j)$ , created through the merger of arcs  $(i, k)$  and  $(k, j)$ , the node  $k$  has to be in the diagonal block.

---

2. There may be nodes whose predecessor updates were not deferred but they are in the successor list of some node in the diagonal block. Unnecessary reading and processing of such predecessor lists may be avoided by keeping track in a bit vector during Phase I and II of those nodes whose predecessor updates have been deferred.

**Algorithm 2 : Blocked Warshall with Predecessors**

```

For each column partition
    (columns  $j_b$  to  $j_e$  inclusive)

/* Phase I: Processing of diagonal block */
For  $j = j_b$  to  $j_e$ 
    For  $i = j_b$  to  $j_e$ 
        If tuple  $\langle i, j \rangle$  exists
            Add succ. list  $j$  to succ. list  $i$ 
            For every  $k$  in succ. list of  $j$   $j_b \leq k \leq j_e$ 
                Add  $i$  to pred. list of  $k$ 

/* Phase II: Processing of off-diagonal rows */
For  $i = 1$  to  $v \wedge i \notin j_b$  to  $j_e$ 
    For  $j = j_b$  to  $j_e$ 
        If tuple  $\langle i, j \rangle$  exists
            Add succ. list  $j$  to succ. list  $i$ 
            For every  $k$  in succ. list of  $j$   $j_b \leq k \leq j_e$ 
                Add  $i$  to pred. list of  $k$ 

/* Phase III: Predecessor Updates
of off-diagonal nodes */
For  $j = 1$  to  $v \wedge j \notin j_b$  to  $j_e$ 
    For  $i = j_b$  to  $j_e$ 
        If tuple  $\langle i, j \rangle$  exists
            Add pred. list  $i$  to pred. list  $j$ 

```

Consider again the matrix in Figure 1, and assume that the column partitions are same as before. Figure 2 shows the order in which various matrix elements would be examined for processing the middle partition. In this figure,  $d$   $i$ 's give the ordering during Phase I,  $s$   $i$ 's during Phase II, and  $p$   $i$ 's during Phase III.

	$s1$	$s2$	$s3$	
	$s4$	$s5$	$s6$	
$p1$	$d1$	$d4$	$d7$	$p7$
$p2$	$d2$	$d5$	$d8$	$p8$
$p3$	$d3$	$d6$	$d9$	$p9$
	$s7$	$s8$	$s9$	
	$s10$	$s11$	$s12$	

**Figure 2.** Processing of Middle Column Partition in Blocked Warshall with Predecessors

The relative order in which the off-diagonal successor lists are updated in Phase II, and the order in which predecessor lists of the off-diagonal nodes are updated in Phase III can be arbitrarily changed without affecting correctness. Further note that if at the end of the computation of closure, the complete and correct predecessor list of each node (reverse closure)

is not required then, during Phase III, the predecessor lists of off-diagonal nodes to the left of the diagonal block (nodes 1 to  $j_b-1$ ) need not be updated, as these predecessor lists are not referenced in further processing. A little thought will convince the reader that in such a situation the predecessor updates during Phase I and II of the algorithm are also not required. Recall that the only function of the predecessor lists is to restrict during Phase II the input of those *off-diagonal* successor lists whose row number is not present in the predecessor list of any of the nodes in the current diagonal block. Clearly, predecessor lists are not required during Phases I. The result of not updating the predecessor lists in Phases I and II is that the predecessor lists of some nodes may be too short in that they miss some nodes whose successor lists have been updated. However, all such missing nodes belong to the diagonal block and they are immaterial for row selections in Phase II.

**Dynamic Partitioning**

The dynamic partitioning works in a way very similar to Blocked Warshall. If the memory becomes full during the diagonal block processing, the size of the partition may be reduced by discarding the successor and predecessor lists corresponding to the last column from the memory. The only difference is that if these lists have been partially updated, we do not have the option of writing back these partial updates and saving some future computation — we will have to discard the updates for the algorithm to correctly compute the closure.

**3.7 Blocked Warren Algorithm**

Consider once again the problem posed at the beginning of the previous sub-section. In the Blocked Warshall algorithm, we have the successor list of  $j$  in memory at the time that we process element  $(i, j)$ , and the successor list of  $i$  has to be read in if required. The problem is that whether this successor list is required for update depends on whether a tuple  $(i, j)$  exists, and this information available only in the successor list of  $i$  which is not in memory. One way to fix this problem was to maintain predecessor lists, as in Blocked Warshall with Predecessors. Another way is to reverse the roles of  $i$  and  $j$ . What if the successor list of  $i$  is in memory when element  $(i, j)$  is processed, so that successor list  $j$  need be read only if it is required to update  $i$ ? Now our objective becomes to minimize such fetches to row  $j$ . In a manner analogous to algorithm 1, this objective suggests a blocking technique wherein a row partition is processed at a time. We can then proceed column-wise within each row partition, fetching  $j$  only once

for all the  $(i,j)$ ,  $(k,j)$ , etc., elements within the partition.

We find that the first precedence constraint is easily satisfied, but the second precedence constraint prevents us from going beyond the diagonal block as we process each row partition in turn. A second pass is required to mop up the rest just as in the ordinary Warren algorithm. The Blocked Warren algorithm can now be written:

**Algorithm 3 : Blocked Warren**

```

/* First Pass */
For each row partition (rows  $i_b$  to  $i_e$  inclusive)
  For  $j = 1$  to  $i_e$ 
    For  $i = i_b$  to  $i_e$ 
      If tuple  $\langle i,j \rangle$  exists
        Add succ. list  $j$  to succ. list  $i$ 

/* Second Pass */
For each row partition (rows  $i_b$  to  $i_e$  inclusive)
  For  $j = i_b$  to  $n$ 
    For  $i = i_b$  to  $i_e$ 
      If tuple  $\langle i,j \rangle$  exists
        Add succ. list  $j$  to succ. list  $i$ 

```

In the second pass, one could remember all the last element in each row examined in the first pass (remember that we evaluated beyond the diagonal element in many cases) and examine the rest. Furthermore, the row partitioning in the second pass need not be same as in the first pass.

Figure 3 shows the order in which the elements of a  $7 \times 7$  matrix will be processed using the Blocked Warren algorithm. The horizontal lines bracket the row partitions and the thick stair-way lines separate the two passes. Notice that the order of computation is significantly different from the straight Warren, straight Warshall, or Blocked Warshall.

1	3	34	35	36	37	38
2	4	39	40	41	42	44
5	8	11	14	17	43	45
6	9	12	15	18	46	48
7	10	13	16	19	47	49
20	22	24	26	28	30	32
21	23	25	27	29	31	33

**Figure 3.** The order of computation in the Blocked Warren algorithm

**Dynamic Partitioning**

In Blocked Warren also, it is possible that the set of successor lists in memory may keep adding on tuples until it is too large to fit in memory any more. In that case, the last successor list can be discarded, and the corresponding row included in the next partition. As in Blocked Warshall, we have a choice about whether the discarded successor list is written back, so that some computing and update may be saved in the processing of the next partition, or whether the discarded list is simply written over in memory, saving I/O but requiring a recomputing of some updates.

**4. PERFORMANCE EVALUATION**

In this section we present the results of simulation experiments that we performed to study the performance of the algorithms presented in Section 4. We first make a few comments on the performance evaluation methodology, and describe the datasets used in the study.

**4.1 Methodology**

Since the publication of [4], the Wisconsin Benchmarks have become the de facto standard for evaluating the performance of database systems and algorithms. The main merit of these benchmarks, in our opinion, is the ability to specify the selectivity for a relational operation. As a consequence, given an initial relation, one can exercise control over the size of the result relation by modifying this selectivity.

Following the same argument, it would be nice to specify some single parameter that would, in conjunction with the size of the starting relation, be a good predictor of the size of the relation that results upon taking a transitive closure. The average degree of a node makes an excellent choice in this regard. It is a property that is easy to determine for any given initial relation, and it is easy to see that the size of the result relation will grow as the average node degree grows.

In a directed graph, one must be concerned with both the out-degree and the in-degree of a node. However, in a random graph, each in-degree for some node must be balanced by an out-degree for some other node, and the average in-degree must be equal to the average out-degree. Therefore, as long as we are dealing only with averages, it does not matter which of the two we consider. Indeed we tried out several datasets in which the in-degree was specified and found that the results obtained were not significantly different from corresponding datasets in which the out-degree was specified. We, therefore, present only

the datasets with the out-degree specified, and the corresponding results. The out-degree of each node is obtained from a uniform distribution between zero and twice the specified mean.

There is one other aspect of a directed graph that we consider important — that is its *locality*, measured as the average length of an arc, where the length of every arc is the absolute difference of its source node number and destination node number. Thus in a graph with high locality, most arcs from a node would connect nearby nodes, with a few arcs that connect far-away nodes. One might expect to find such a property, for example, in a database of inter-city bus-routes. The length of each arc is obtained from an exponential distribution with a specified mean. (Nodes are assumed to be numbered modulo the maximum node number so that an arc from the last node to the first would be considered of length 1 and so on). Thus, low values for this mean length cause high locality, and high values result in a uniform arc distribution (low locality). Notice that a uniform graph will have an average arc-length equal to one-quarter the number of nodes in the graph, and given our exponential generation model, it is not possible to consistently obtain a greater average arc length.

In addition to the random graphs discussed thus far, we also considered trees and inverted trees. For these kinds of graphs, we specified the average branching factor, which happens to be the same as the average out-degree for a tree and average in-degree for an inverted tree, excluding the leaf nodes. The actual branching factor for each node was once again obtained from a uniform distribution. Locality is not a parameter for these graphs.

Our experiments were performed with the databases shown in Table 1. The random tuples generated were sorted and duplicates eliminated in a post-processing step. In most cases, this duplicate elimination did not make a significant difference. However, when the locality was very high (very low average arc-length specified), there was considerable duplication of arcs between immediately neighboring nodes. Therefore the actual degree of each node became considerably less than the nominal degree specified. Also, the actual average arc-length was substantially greater (since the duplicates eliminated were predominantly short arcs) than the nominal arc-length specified.

The performance metric used was the total I/O generated by the algorithms in kilobytes<sup>3</sup>. We did

not collect statistics on CPU costs, as for large database relations, the total cost would be dominated by the I/O costs. We also did not model buffering, the justification being that when the size of the memory is a small fraction of the result relation size, it is unlikely that significant useful information will be found in the memory itself without having to access disk. On the other hand, if the size of the memory approaches the size of the final relation size (or is greater than it), then buffering is an important consideration, but also, the I/O costs drop (and are actually the same for any algorithm if the memory can hold the entire result relation — input equal to the initial relation size, and output equal to the final relation size) so that a model that ignores compute costs is not interesting in the first place. In short, our cost models are appropriate for databases which are large relative to the size of the memory available.

#### 4.2 Experiments

Table 2 shows the values of the simulation parameters used in the performance experiments, unless otherwise stated:

**Table 2. Simulation Parameters**

Parameter	Value
Memory Size (M)	500 Kilo Bytes
Tuple Size	100 Bytes
Size of the Key Fields	10 Bytes

The size of the memory was chosen so that it was approximately one-tenth of the size of the final relation<sup>4</sup>. All of the databases discussed here were generated to produce a result relation approximately equal in size (5 Megabytes), so that we could see the effect of using different types of database structures. A tuple size of 100 Bytes was used throughout. It should be evident that a change in the tuple size automatically causes a change in the size of the relations that have to be stored and is equivalent to an inverse change in the size of the memory. We varied memory size when required rather than tuple size. The size of the key fields is important only for Warshall with Predecessors, where the predecessor

3. We also collected statistics in terms of I/O blocks, but these results have not been presented since they exhibited similar trends overall yet could have specific values manipulated by altering the disk placement and blocking strategy.

4. We performed some preliminary experiments with larger memory size and larger result relations, but similar trends were observed. We resorted to smaller memory size to keep the simulation times reasonable.



**Table 1. Synthetic Databases**

Type	Name	Number of Nodes	Number of Arcs	Nominal Out-Degree	Nominal Arc-Length	Avg. Arc Length	Arcs in Result
Uniformly Random	u.1	2700	2685	1	large	564.2	49873
	u.10	230	2165	10	large	47.4	50601
Random with High Locality	h.1	2612	2596	1	1	1.6	46127
	h.10	390	1603	10	1	2.0	50092
Random with Medium Locality	m.1	7200	7152	1	10	10.5	50036
	m.10	230	2031	10	10	11.4	50601
Tree	t.1	1720	1731	1 <sup>a</sup>			50282
	t.10	1200	12196	10 <sup>a</sup>			50176
Inverted Tree	it.1	1720	1731	1 <sup>a</sup>			50282
	it.10	1200	12196	10 <sup>a</sup>			50176

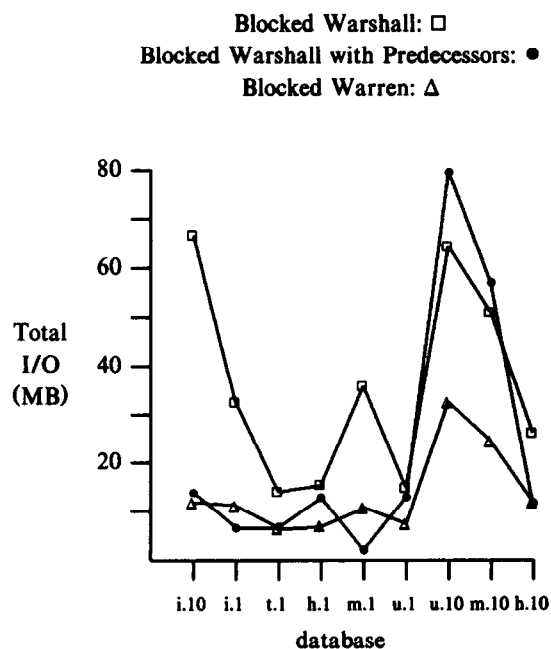
a: Average Out-degree of non-leaf nodes.

b: Average In-degree of non-leaf nodes.

lists consist of abbreviated tuples with only the key fields, while the successor lists as usual carry the entire tuple information with them. The larger the fraction of the tuple occupied by the key fields, the larger the overhead for storing and manipulating predecessor lists, and hence the worse the performance of Warshall with Predecessors.

**4.2.1 Experiment 1: Comparative Performance of the Algorithms**

Figure 4a shows the comparative performance the three algorithms for nine databases<sup>5</sup>. Figures 4b and 4c show the total number of bytes read from disk and the total number of bytes written to disk respectively. These numbers add up to give the total numbers plotted in Figure 4a. The reduction in reading costs for Warshall on account of keeping predecessor lists is evident from Figure 4b. On the other hand, no benefit accrues in the cost of writing on this account, and in fact some overhead results.



**Figure 4a. Comparative performance of the three algorithms (Total I/O)**

Overall, Blocked Warren performs better than the other two algorithms, especially as the degree of the graph increases and locality is absent. The reason for this behavior is that each successor list is likely to be updated more often, the higher the degree of the graph. In the absence of locality, these updates will all take place due to interaction with many different nodes which are all in different partitions. In the case of the two Warshall algorithms, a successor list is written back after being updated once for each partition processed. In Warren, the list is written back only once, when the partition to which this list belongs is processed. The lower writing costs of Warren, therefore, are evident.

5. We have not presented the results for t.10. Notice that in such a database, there would be a path to every node from the root node, and the successor list of the root node requires a disproportionately large amount of storage, and exhibits an astonishing growth rate. In fact the successor list for the root of the tree in t.10 does not even fit in the entire (half a megabyte of) memory that we have.

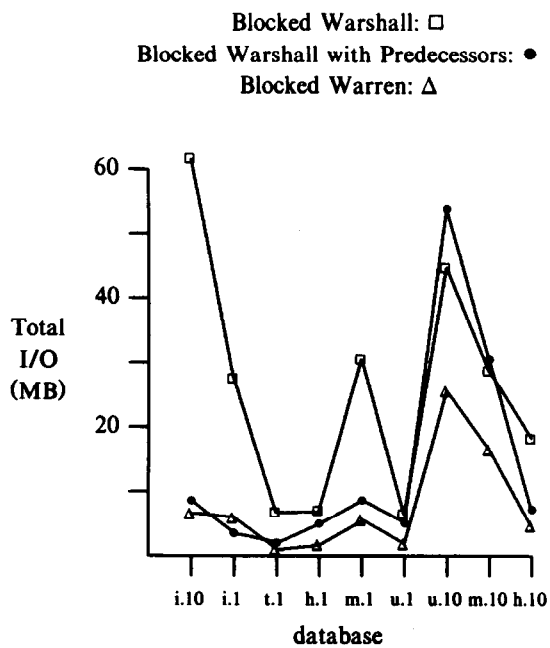


Figure 4b. Comparative performance of the three algorithms (Reads only)

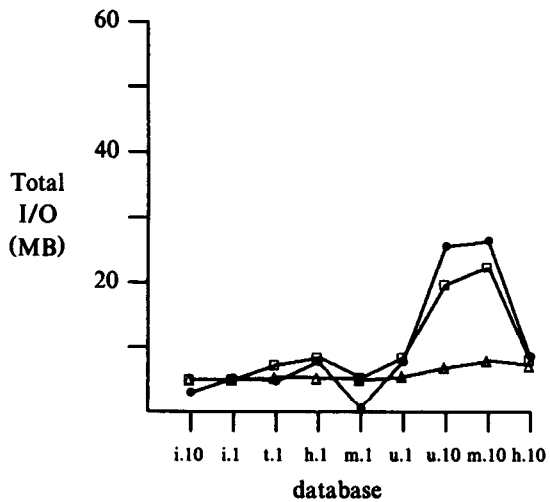


Figure 4c. Comparative performance of the three algorithms (Writes only)

Most of the reading costs is not in reading the partition to be processed but in reading the successor lists that interact with this partition (this is an  $n$  versus  $n$ -squared type of situation). Therefore the fact that Blocked Warren reads each partition twice is not a major added cost. On the other hand, the unnecessary reads in Blocked Warshall, and the overhead of the predecessors in Blocked Warshall with Predecessors, both are significant additional costs. Therefore, Blocked Warren outperforms these algorithms in the read cost as well.

We varied the memory size for all of the above cases and found that the results obtained were similar. These results are not presented here for conciseness. The only point we wish to make is that the performance of Blocked Warshall is extremely sensitive to the memory size (primarily on account of the unnecessary reads), whereas the performance of the other two algorithms is only moderately sensitive to the memory size. No significant changes in the relative performances of these algorithms is expected for any memory size that is much smaller than the final relation size.

#### 4.2.2 Experiment 2: Comparison with an Iterative Algorithm

In [14], the performance of the semi-naive algorithm has been compared to the performance of a logarithmic algorithm, and the semi-naive and several logarithmic algorithms have been compared in [8]. The logarithmic algorithms were found to perform better than the semi-naive algorithm in both [14] and [8]. We, therefore, selected the logarithmic algorithm as the best iterative algorithm to compare it against our direct algorithms. We simulated the optimized version of this algorithm as presented in [10]. Because of the superior performance of hash-based join algorithms [6], the joins were performed using an idealized hash-based join algorithm. The major problem with the hash-based join algorithms is that of guaranteeing that a chosen partitioning of hash values will result in buckets that will fit in memory, and many strategies such as bucket tuning and recursive repartitioning have been proposed to deal with this problem [6]. In our simulation, we assumed perfect partitioning of relations so that the partitions never overflow and never have to be adjusted. Thus the numbers presented below for the logarithmic algorithm represent a lower bound on the I/O cost that is the best that the cleverest partitioning scheme could hope to achieve. As in the simulation of the direct algorithms, we assumed that there was no buffering.

Figure 5 shows the relative performance of Blocked Warren against the lower bound for the logarithmic algorithm. It is clear that the direct algorithms are considerably superior to the iterative ones. In fact, for the random graphs with degree 10, the performance of the logarithmic algorithm was so poor that the costs could not reasonably be plotted on the same linear scale as the costs for the direct algorithms. We present these results in Table 3.

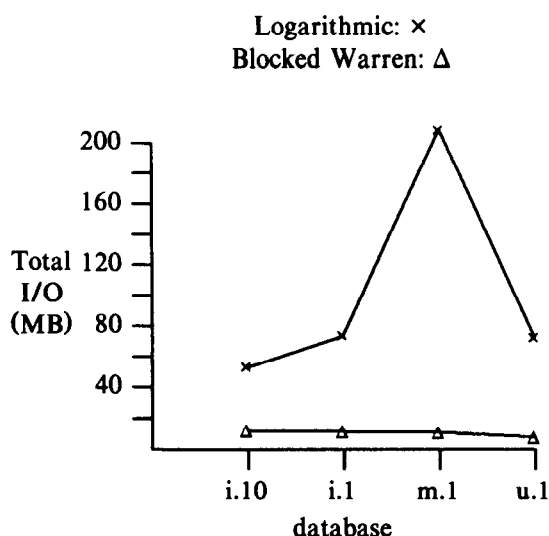


Figure 5. Comparative performance of Direct and Iterative algorithms

Table 3. Comparison of Iterative and Direct Algorithm I/O Costs

Data Base	Blocked Warren	Logarithmic	Improve ment
h.10	11736	9255161	789
m.10	24383	6408957	263
u.10	32395	9293838	287

## 5. CONCLUSIONS

We have presented three algorithms to compute the transitive closure of a database relation. These algorithms are "direct" in the sense that they rely upon the special structure of the transitive closure problem rather than solving a general recursion. All three algorithms consistently out-performed the well-regarded logarithmic algorithm for computing transitive closure, for each of several databases studied. In many cases we were able to show an improvement several orders of magnitude. Of the three algorithms presented, Blocked Warren seemed, by and large, to do better than the others. Blocked Warshall with Predecessors could be an option worth considering for sparse graphs with high locality.

In our simulation experiments, we assumed that the memory size is small compared to the result relation size. In a recent paper [10], a straightforward implementation of Warren was compared against the logarithmic algorithm and was found to do better when the memory size was not much smaller than the final relation size. It is easy to show that the Blocked Warren algorithm that we propose will always

perform at least as well as the straightforward Warren used in [10]. Therefore, we expect that Blocked Warren will perform better than iterative algorithms even when the memory size is large. Moreover, the performance of Blocked Warshall improves rapidly with increasing memory size, and we expect that it too will perform well with large memory size.

We may have given the impression that the algorithms given in this paper can only be used to determine reachability in a graph (whether a path exists between two nodes). However, if a problem obeys the path algebra developed by Carre [5] (and many important path problems such as the shortest path, the maximum reliability path, the critical path etc. fall in that category), then such computations can be performed with obvious minor modifications to these algorithms. Let us consider the computation of the shortest path in a graph. Now, at the time of adding  $j$  to the successor list of  $i$  as a result combining the two arcs  $(i,k)$  and  $(k,j)$ , the distance attribute of  $(i,j)$  is also computed (as the sum of the distance attributes of  $(i,k)$  and  $(k,j)$  respectively) and stored along with  $j$  in the successor list of  $i$ . If  $j$  is already in the successor list of  $i$ , the duplicate is eliminated by retaining the tuple with the smaller value for the distance.

## REFERENCES

- [1] R. Agrawal, Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries, *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590.
- [2] R. Agrawal and H. V. Jagadish, Direct Algorithms for Computing the Transitive Closure of Database Relations, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, 1987.
- [3] F. Bancilhon, Naive Evaluation of Recursively Defined Relations, Tech. Rept. DB-004-85, MCC, Austin, Texas, 1985.
- [4] D. Bitton, D. J. DeWitt and C. Turbyfill, Benchmarking Database Systems - A Systematic Approach, *Proc. 9th Int'l Conf. Very Large Data Bases*, Florence, Italy, Oct. 1983.
- [5] B. Carre, *Graphs and Networks*, Clarendon Press, Oxford, 1978.

- [6] D. J. DeWitt and R. H. Gerber, Multiprocessor Hash-Based Join Algorithms, *Proc. 11th Int'l Conf. Very Large Data Bases*, Stockholm, Sweden, Aug. 1985, 151-164.
- [7] A. Guttman, New Features for Relational Database Systems to Support CAD Applications, Computer Sciences Dept., Univ. California, Berkeley, June 1984. Ph.D. Dissertation.
- [8] Y. E. Ioannidis, On the Computation of the Transitive Closure of Relational Operators, *Proc. 12th Int'l Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, 403-411.
- [9] H. V. Jagadish, R. Agrawal and L. Ness, A Study of Transitive Closure as a Recursion Mechanism, *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Francisco, California, May 1987.
- [10] H. Lu, K. Mikkilineni and J. P. Richardson, Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation, *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 112-119.
- [11] A. Rosenthal, S. Heiler, U. Dayal and F. Manola, Traversal Recursion: A Practical Approach to Supporting Recursive Applications, *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 166-176.
- [12] L. Schmitz, An Improved Transitive Closure Algorithm, *Computing* 30, (1983), 359-371.
- [13] C. P. Schnorr, An Algorithm for Transitive Closure with Linear Expected Time, *SIAM J. Computing* 7, 2 (May 1978), 127-133.
- [14] P. Valduriez and H. Boral, Evaluation of Recursive Queries Using Join Indices, *Proc. 1st Int'l Conf. Expert Database Systems*, Charleston, South Carolina, April 1986, 197-208.
- [15] H. S. Warren, A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations, *Commun. ACM* 18, 4 (April 1975), 218-220.
- [16] S. Warshall, A Theorem on Boolean Matrices, *J. ACM* 9, 1 (Jan. 1962), 11-12.
- [17] M. M. Zloof, Query-By-Example: Operations on the Transitive Closure, RC 5526, IBM,

Yorktown Hts, New York, 1975.