A PERFORMANCE STUDY OF TRANSITIVE CLOSURE ALGORITHMS †

Shaul Dar +

Raghu Ramakrishnan *

ABSTRACT

We present a comprehensive performance evaluation of transitive closure (reachability) algorithms for databases. The study is based upon careful implementations of the algorithms, measures page I/O, and covers algorithms for full transitive closure as well as *partial* transitive closure (finding all successors of each node in a set of given source nodes). We examine a wide range of acyclic graphs with varying density and "locality" of arcs in the graph. We also consider query parameters such as the selectivity of the query, and system parameters such as the buffer size and the page and successor list replacement policies. We show that significant cost tradeoffs exist between the algorithms in this spectrum and identify the factors that influence the performance of the algorithms.

An important aspect of our work is that we measure a number of different cost metrics, giving us a good understanding of the predictive power of these metrics with respect to I/O cost. This is especially significant since metrics such as number of tuples generated or number of successor list operations have been widely used to compare transitive closure algorithms in the literature. Our results strongly suggest that these other metrics cannot be reliably used to predict I/O cost of transitive closure evaluation.

1. Introduction

We present a comprehensive performance evaluation of transitive closure (reachability) algorithms for databases. The novel aspects of our work are:

- We study full transitive closure as well as partial transitive closure [18], and our results include the first comparisons of page I/O cost for partial transitive closure algorithms. We compare the performance of the algorithms over a wide range of graphs and queries and identify the factors that influence the performance of the algorithms in this spectrum.
- We formulate and implement the algorithms in a uniform framework as variations of a single base algorithm, highlighting their similarities and differences. The implementation framework is detailed and includes clustering of data on disk and in memory, indexing, and buffer management.
- In addition to measuring page I/O (which is widely accepted as the dominant cost measure in database query evaluation), we measure other cost metrics such as the number of tuples generated and tuple and successor list I/O. These metrics were used in many previous studies of transitive closure and recursive query processing. However, our results demonstrate that these cost metrics cannot be reliably used for predicting the page I/O cost.
- We propose a new qualitative model of DAG workloads with parameters ("height" and "width") that can be measured in a single traversal of the graph.

We briefly comment on these points below. We have chosen what we consider to be the best set of algorithms based on previous performance studies, which we review in the related work section. The candidate algorithms are the BTC algorithm [12], the Hybrid algorithm [2], the BFS algorithm [18], the Search algorithm (see e.g. [15]), the Spanning Tree algorithm [6] and the Compute_Tree algorithm [15].

We study the effect of varying graph parameters such as the density and "locality" of arcs in the graph, and we examine a wide range of acyclic graphs. Our decision to study acyclic graphs is based on the well known observation that, given a cyclic graph, an acyclic *condensation graph* (in which strongly connected components are merged) can be computed cheaply in comparison to the cost of computing the closure of the condensation graph (see e.g. [28]). We also consider query parameters such as the selectivity of the query, and system parameters such as the buffer size and the page and successor list replacement policies. We show that significant cost tradeoffs exist between the algorithms in this spectrum.

We have developed a uniform framework for describing and implementing the algorithms that we study. These algorithms

† This work was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and a grant from IBM.

+ AT&T Bell Labs, 600 Mountain Ave., Murray Hill, New Jersey 07974. Work done partially while the author was at the University of Wisconsin.

* Computer Science Dept., University of Wisconsin, 1210 W. Dayton St., Madison, Wisconsin 53706.

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 ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

were developed independently by several researchers and were presented in different ways. In spite of the apparent differences, we were able to formulate all the algorithms as variations of a single, two-phase algorithm, based on manipulation of successor lists. In the restructuring phase, which is common to all the algorithms, the tuples of the input relation are converted into successor lists in memory. For selection queries the "magic" subgraph comprising the nodes and edges reachable from the specified source nodes is identified during this phase. In the computation phase, which is different for each algorithm, the successor lists are expanded and then written out to disk. The uniform framework makes it easier to understand the essential similarities and differences between the algorithms, and their effect on their relative performance. Further, it ensures that our implementation of the algorithms is consistent.

In measuring the performance of the algorithms we used page I/O as the primary cost measure. However, we also obtained extensive measurements in terms of several other metrics, some of which have been widely used for comparing algorithms for transitive closure and recursive query processing (see Section 7). This has several benefits. First, we have identified some important factors that affect the relative performance of the algorithms (e.g., "marking utilization", "selection efficiency"). Second, we are able to show that our results are consistent with earlier comparisons of partial transitive closure algorithms that use cost metrics other than page I/O. Finally, and perhaps most importantly, the results strongly indicate that widely used cost metrics such as the number of tuples generated or the number of successor list operations do *not* predict page I/O cost reliably.

We also present a new model of DAG workloads with parameters that can be measured in the first pass of our two-pass implementation of the algorithms. There is a qualitative correlation between the "shape" of a DAG as measured by this model and the relative performance of some of the algorithms. While our model is not sophisticated enough to allow a query optimizer to choose the "best" algorithm for the second phase, we believe it is a useful first step in that direction.

The rest of this paper is organized as follows. In Section 2 we provide some background and notation. In Section 3 we describe the algorithms that we study, and in Section 4 we outline the implementation framework. In Section 5 we present the experimental setup. In particular, we describe the input graphs used in our study using a novel model for characterizing directed acyclic graphs. In Section 6 we present and analyze the results of our experiments. In Section 7 we offer our insight on the evaluation methodology for transitive closure algorithms. In Section 8 we discuss related work, and in Section 9 we present our conclusions.

2. Background and Notation

Let G be a directed graph with n nodes and e arcs. We denote the transitive closure of G by $TC(G)$. We can cast the transitive closure computation as the expansion of the *successor list* of each source node x, as follows:

1. Initially, the *immediate* successor list of a node x is given by $S_x = \{ y \mid (x,y) \in G \}$. The nodes in the immediate successor list of x are called the *children* of x.
2. After the computation of the closure, the *full* successor list of x is given by $S_x = \{ y \mid (x,y) \in TC(G) \}$. The nodes in the full successor list of x are called the *successors* of x.

In complete (full) transitive closure computation (CTC), we need to find the full successor list of each node x in G. In partial transitive closure computation (PTC), we need to find the full successor list of each node x in a set S of source nodes. If S contains exactly one node, we say that the PTC is *single-source*. Otherwise we say that the PTC is *multi-source*. (Our definition corresponds to *strong* multi-source PTC computation in [18].) Given a PTC query Q specifying a set S of source nodes, we refer to the subgraph of G containing those nodes and edges of G reachable from some node $s \in S$ as the *magic* subgraph of G wrt. Q, and denote it by G_m.

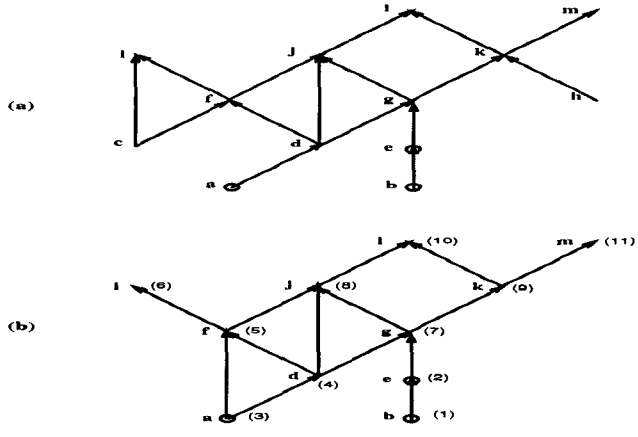

Figure 1. Magic (Reachable) Graph

Given a directed acyclic graph (DAG) G, a *topological order* on the nodes of G is an ordering $\prec_t$, such that if $(x,y) \in G$, then $x \prec_t y$.

As an example, consider the DAG in Figure 1 (a) and the set of source nodes, marked with a circle, $S = \{a,b,e\}$. The magic subgraph for this selection is shown in Figure 1 (b). Next to each node in the magic graph, in parenthesis, appears its numbering (also called *rank*) in a possible topological ordering.

3. The Algorithms

In this section we provide a brief overview of the algorithms that we study. The description should be sufficient to understand the performance results of Section 6. The reader can find more detailed descriptions in [7] or in the original papers.

3.1 The BTC Algorithm

Given a DAG G, the BTC algorithm computes a topological order of the nodes in G and then expands the successor lists of the nodes in G in reverse topological order. The processing of a node involves reading the successor lists of its children and merging them with its own list (this is called the *immediate successor optimization* in [2]). For example, in the graph of Figure 1 (a) the processing of node d involves reading S_f and S_g, which at this point are fully expanded, and merging them to create S_d. The children of a node are considered in topological

order, to avoid unnecessary successor list unions [8, 10]. For example, when processing node d in the graph of Figure 1 (a) node f is considered before node j, and the algorithm avoids the union of S_j with S_d. Although j is a child (immediate successor) of d, it is also an indirect successor of d through node f (and g). Since node f is processed before node d, S_f already contains all the nodes in S_j at the point where node d is processed, and the union of S_f with S_d obviates the union of S_j with S_d. The BTC algorithm marks child nodes that should not be processed, such as node j. Hence, we refer to this optimization as the *marking optimization* [2]. As shown in [10, 17], the marking optimization on a topologically sorted graph is equivalent to a *transitive reduction* of the input graph [4]. In other words, if we remove from the original graph all marked arcs, we end up with the minimal graph that has an identical closure to the original graph.

3.2 The Hybrid Algorithm

In the Hybrid algorithm [2], as in the Direct algorithms [3], successor lists are read into memory and expanded one block at a time in order to take advantage of locality and reduce I/O (see Figure 2). The block of successor lists in memory, shaded in Figure 2, is called the *diagonal block*. A successor list is called a *diagonal list* if it is part of the diagonal block, and an *off-diagonal list* otherwise. The union of two diagonal lists requires no I/O since both lists are in memory. On the other hand, the union of a diagonal list with an off-diagonal list requires I/O to bring the off-diagonal list into memory. However, when an off-diagonal list j is brought into memory (displacing a previous off-diagonal list), it is joined with all diagonal lists i_k such that j is a child of i_k. Figure 2 shows schematically an off-diagonal list that may be unioned with three diagonal lists. Thus, blocking allows several successor list unions to be performed at the cost of a single I/O.

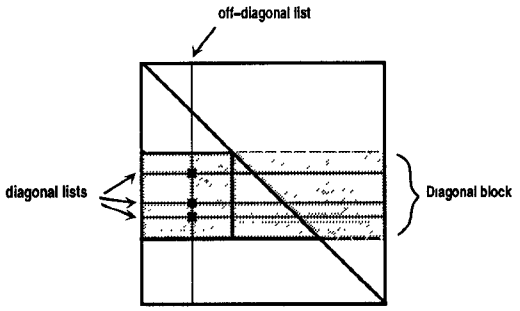

Figure 2. Blocking

3.3 The BFS Algorithm

Jiang [18] made the following observation. Given a multi-source query with a set S of source nodes, if a node j has a single parent i and j is not in S, then there is no need to compute the successor list of j. Instead, node j can be *reduced* to a sink node by making its children (immediate successors) into children of i and deleting the outgoing arcs of node j. We call this technique the *single-parent optimization*. As an example, consider the magic graph in Figure 1 (b) with the set of source nodes $S = \{a,b,e\}$. Nodes d and k are single-parent and can be reduced. Nodes f, g and j, the children of node d, become children of node a, the parent of d, and node d becomes a sink node. Similarly, nodes l and m, the children of node g, are adopted by node g, the parent of k. The resulting graph is shown in Figure 3. Note that node e is not reduced since it is in S.

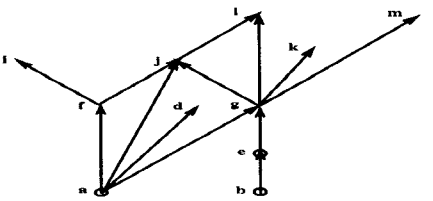

Figure 3. The Single-Parent Optimization

3.4 The Search Algorithm

When the number of source nodes in the query (i.e., the size of S) is small, the extra cost of a topological sort and of expanding non-source nodes in the magic graph may not be justified. A simple search of the graph, starting from the source nodes and expanding only their successor lists, may be cheaper. We call such an algorithm a *search* algorithm (following [14]). With a search algorithm, a multi-source query with k source nodes is essentially treated as k single-source queries.

3.5 The Spanning Tree Algorithm

Jakobsson [14] and Dar and Jagadish [6] independently suggested that if the closure computation makes use of *successor trees* instead of "flat" successor lists then redundant work and unnecessary I/O may be avoided. Consider the graph of Figure 1 again. Assume that the processing of node d is done by first merging S_f with S_d and then merging S_g with S_d. The common part of S_f and S_g, namely nodes j and l, will be inserted to S_d after the first union. Yet these nodes will be redundantly read from S_g as part of the second union. If successor information is represented by successor spanning trees then the union of S_g with S_d may be done more efficiently. After finding that j is already in S_d there is no need to read any successors of j in S_g, since they must already be present in S_d. (The successor tree S_g contains structural information indicating that node l is the child of node j).

3.6 The Compute_Tree Algorithm

In [15], Jakobsson presented a spanning tree algorithm tailored to partial transitive closure. The algorithm, called *Compute_Tree*, differs from the CTC spanning tree algorithm of [14] in two important ways:

- The spanning trees are built with respect to the arc-reversed graph, i.e., the algorithm uses *predecessor trees* rather than successor trees.
- A predecessor tree for a node x contains only a subset of the predecessor nodes of x. These nodes are called *special* nodes (with respect to x). A node is special if it is either a source node or the nearest common ancestor of at least two source nodes that are unrelated (that is, there is no path between them). Because a predecessor tree only stores special nodes, the size of such a tree is at most twice the size of the set S, i.e., the number of source nodes specified in the query [15].

As an example, consider the magic graph in Figure 1 (b). Four possible special-node predecessor trees, for nodes f, g, j and k, are shown in Figure 4 (a), (b), (c) and (d).

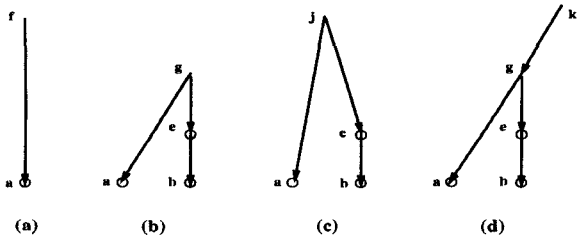

Figure 4. Special-Node Predecessor Trees

4. Implementation

The execution of all the algorithms (except the Search algorithm) is divided into two phases.

1. **Restructuring phase**: This phase is common to all of the algorithms. During the restructuring phase the nodes are topologically sorted. At the same time, the tuples of the input relation are converted into *successor list format* resulting in a more compact representation (see [7]). For selection queries, the magic subgraph is identified during this phase, and the topological sort and construction of successor lists are done with respect to this graph.

2. **Computation phase**: This phase is different for each algorithm. It involves expanding the successor lists and then writing the expanded lists out to disk. For selection queries, only the expanded lists of the query source nodes are written out.

We assume that the corresponding relation is stored on disk as a set of tuples clustered on the source attribute. We also assume the existence of a clustered index on the source attribute. The in-memory representation of successor lists uses fixed-length blocks, with the last immediate successor in each list designated by negating its value. Our implementation of the algorithms was very detailed and included inter- and intra- successor list clustering and a choice of several page and set replacement policies. Additional details about the data structures used to represent the graph on disk and in memory and the management of these data structures can be found in [7].

4.1 Uniform Implementation Framework

We observed that in spite of apparent differences between the candidate algorithms, they can all be naturally formulated and implemented as manipulations of successor lists. Consequently, we implemented the algorithms in the study as variations of one basic algorithm, namely the BTC algorithm. We denote the implementation of the basic algorithm as *BTC*. The important variation to the processing of successor lists in the candidate algorithms are as follows:

- The Hybrid algorithm uses blocking of successor lists. A block of successor lists at a time is read into memory and expanded. If the memory is filled up during expansion, the current block size is decreased by discarding one or more pages from the current block (this is called *dynamic reblocking* in [2, 3].) We denote our implementation of the Hybrid algorithm by *HYB*.

- The BFS algorithm uses the single-parent optimization. The successor lists of single-parent nodes are not expanded. Instead, the children of such nodes are adopted by their parents. We denote our implementation of the BFS algorithm by *BJ*.

- The Search algorithm computes a single-source PTC for each source node. The immediate successor optimization is not used — the successor list of a node x is unioned with the *immediate* successor list of every node y that is a *successor* of x. (With the other algorithms, which use the immediate successor optimization, the successor list of a node (x) is unioned with the *full* successor list of every node y that is an *immediate* successor of x.) We have implemented the Search algorithm by extending the preprocessing phase to expand the successor lists of the source nodes specified in the query. The computation phase is no longer needed. We denote our implementation of the Search algorithm by *SRCH*.

- The Spanning Tree algorithm uses successor trees, which are essentially successor lists containing additional structural information. The union of successor trees of nodes x and y uses the structural information to reduce the number of arcs fetched from the tree of y and the number of duplicate arcs generated for the tree of x. Successor spanning trees are represented by storing each parent (internal node) once, followed by a list of its children. Parent nodes are distinguished by negating their values. We denote our implementation of the Spanning Tree algorithm by *SPN*.

- The Compute_Tree algorithm uses predecessor trees. These trees contain only those predecessor nodes that are "special" with respect to the root of the tree. The union of the trees of nodes x and y is done in bottom up fashion, copying from the tree of y into the tree of x only those nodes that are special with respect to x.

  Given a PTC query with a set S of source nodes, the Compute_Tree algorithm requires the creation (in the preprocessing phase) of immediate predecessor lists for every node in the magic graph G_m. The identification of the magic graph G_m requires a search forward from the nodes in S. This can be done efficiently if the input relation is clustered and indexed on the source attribute. At the same time, finding the immediate predecessors of a each node x in G_m can me done efficiently if the input relation is clustered and indexed on the destination attribute. Thus, we have two conflicting demands. We have developed two implementations of the Compute_Tree algorithm. The first implementation, denoted *JKB*, assumes that the input relation is clustered and indexed on the source attribute. The second implementation, denoted *JKB2*, assumes a dual representation of the graph in two relations. The graph relation is clustered and indexed on the source attribute. The inverse graph relation is clustered and indexed on the destination attribute (We discuss this issue in detail in [7].)

Our implementations are not identical to the original descriptions of the candidate algorithms; however, we believe that we capture the important ideas underlying these algorithms. Moreover, our implementations are more detailed than any previous performance study we are aware of.

5. Experimental setup

The parameters of each experiment in our study are described in the following subsections.

5.1 System Parameters

The page size in our experiments is 2048 bytes. The input relation tuples are 8 bytes long (two integers). Hence, in the relation format 256 tuples may be stored on a page. After conversion to successor list format in the restructuring phase 450 successor may be stored on each page. (A successor list page is divided into 30 blocks, each holding up to 15 successor nodes.) The configuration of the system for each experiment is determined by the size of the buffer pool (M) and the list and page replacement policies. The buffer pool size is varied between 10, 20, and 50 pages. A *list replacement policy* is used when a successor list expands to the point where at least one of the other lists on the page must be moved to a new page (i.e., the page must be split). A *page replacement policy* is used to select a page to remove from the buffer pool when it is full. By and large, the choice of page and list replacement policies had a secondary effect, and we do not discuss it further (see [7] for details). The results in Section 6 reflect the best combination of list and page replacement policies for a given query and buffer size.

5.2 Query Parameters

Synthetic graphs with varying characteristics are used to explore the performance of the algorithms (see e.g. [1-3, 12, 17, 18]). The parameters used to guide the graph generation process are the number of nodes in the graph (n), the average out-degree (F), and the graph locality (l). The actual out degree of each node is chosen using a uniform distribution between 0 and $2F$. To create a DAG with locality l arcs going out of a node i are restricted to go to higher numbered nodes in the range $[i+1, \min(i+l, n)]$. This definition of locality, which is similar to that of [1, 12], makes it a property of the graph generation routine and not a property of the resulting graph itself. Hence, we refer to the parameter l as the *generation locality*. We have also developed other definitions of locality that do characterize a graph, independently of how it is generated, and we present those in Section 5.3.

For selection queries, the number of source nodes, i.e., the size of the set S, is used to control the selectivity of the query. We denote this value by s. The query parameter space for the experiments is shown in Table 1. We generated 5 graphs of each family (n, F, and l). In addition, for selection queries, we repeated each experiment 5 times, with a different set S of source nodes. The results presented below show the average of these experiments.

Parameter	Symbol	Values
Number of nodes	n	2000
Average out degree	F	2, 5, 20, 50
Generation locality	l	20, 200, 2000
Selectivity	s	2, 5, 20, 200 500, 1000, 2000

TABLE 1. Query Parameters

5.3 Derived Graph Parameters

We now propose some novel statistics that we use to characterize the directed acyclic graphs (DAGS) in our study. These statistics of the input graph (or, in case of selection, the magic graph) are collected as the graph is being scanned during the restructuring phase. In Section 6 we show how this characterization can assist us in making an intelligent choice of which algorithm to employ for the computation phase of the closure.

Definitions (Node Level, Arc Locality)

Given a DAG G, we define the *node level* of each node $i \in G$ as

$$level(i) = \begin{cases} 0 & \text{if i is a sink node} \\ 1 + \max\limits_{(i,j) \in G} level(j) & \text{otherwise} \end{cases}$$

(Similar definitions are given in [11, 16]).

We define the *arc locality* of each arc $(i,j) \in G$ as

$$locality(i,j) = level(i) - level(j)$$

The locality of an arc (i,j) intuitively indicates the "distance" that the arc spans between i and j. This distance effects the cost of processing the arc, i.e., merging S_j with S_i. Because successor lists are expanded in reverse topological order, the probability that S_j is in memory when S_i is expanded is higher if the distance between i and j is small (the arc (i,j) has *high* locality). The processing of an arc (i,j) does not require a union of the two successor lists, however, if the arc is *marked*, which is the case if an alternative path from i to j is found in G. Hence, the locality of *unmarked* arcs is a better indicator than the locality of all arcs of the probability than a union between S_i and S_j will require an I/O operation to bring S_j into memory. As noted in Section 3.1, when children are expanded in topological order, an arc (i,j) will be marked during the expansion of S_i if and only if (i,j) is in the *transitive reduction* of G. (Note that the transitive reduction of DAG is unique [4].) This discussion motivates the following definitions.

Definition (Redundant Arcs)

Let G be a DAG, and let $TR(G)$ be the transitive reduction of G. We say that an arc $(i,j) \in G$ is *redundant* if $(i,j) \notin TR(G)$. Otherwise, we say that (i,j) is *irredundant*.

Definitions (Height and Width of a DAG)

Given a DAG G, we define the *height* of G, denoted by $H(G)$, as

$$H(G) = \underset{i \in G}{\text{AVG}}\ level(i)$$

Let $|G|$ be the number of arcs in G. We define the *width* of G, denoted by $W(G)$, as

$$W(G) = \frac{|G|}{H(G)}$$

Using these definitions, an arbitrary DAG G is mapped into a *rectangle model*, with height $H(G)$ and width $W(G)$. This is illustrated schematically in Figure 5 (a).

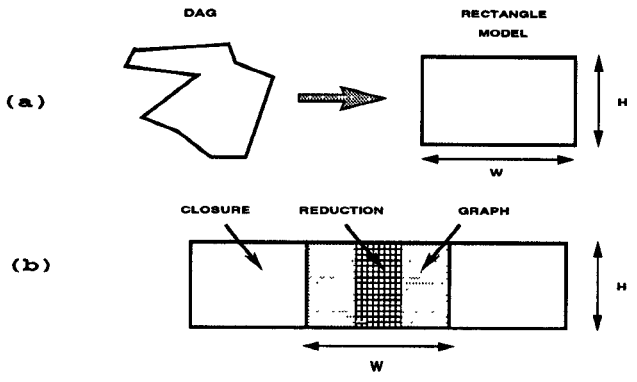

Figure 5. A Rectangle Model for DAGS

Theorem 1 shows an interesting property of this rectangle model.

Theorem 1

Let G be a DAG. Let $TR(G)$ be the transitive reduction of G, and let $TC(G)$ be the transitive closure of G. Then

1. $H(G) = H(TR(G)) = H(TC(G))$
2. $W(TR(G)) \leq W(G) \leq W(TC(G))$ ■

The relationship between G, $TR(G)$ and $TC(G)$ in the rectangle model is illustrated in Figure 5 (b). Intuitively, we suggest that the height of a DAG G is an indication of the "shallowness" of G, while the width of G is an indication of the "degree of redundancy" in G. A high value of $H(G)$ implies that paths in G tend to be long. A high value of $W(G)$ implies that many redundant paths exists in G. In Section 6, we suggest that the rectangle model may be useful in understanding the cost of computing the transitive closure of G. Theorem 2 states that this model can be computed "cheaply" using a single scan of G. In particular, we can compute the rectangle model of G during the restructuring phase at no additional cost.

Theorem 2

Let G be a DAG. We can compute $H(G)$ and $W(G)$ using a DFS traversal of G. ■

Table 2 characterizes the DAGS used in our study. For each DAG, we give its generation parameters F and l (the number of nodes N was 2000 for all the graphs). We then show the resulting number of arcs in the graph [1], the maximum node level, the height and width of the graph according to the rectangle model, the average locality of all arcs and of the irredundant arcs, and finally, the size of the transitive closure of the graph.

We observe that as the out-degree is increased or the generation locality is decreased (higher values of F, lower values of l), the graphs become deeper (higher H and maximum node level). We also note that the average locality of irredundant arcs is much lower than the average locality of all arcs. This emphasizes the importance of the marking optimization: not only does it reduce the number of successor list unions, it also avoids those unions that are likely to be *expensive*, that is, those with high probability of requiring an I/O to be performed.

1. The reader may note that the number of arcs is often less than $N \times F$. There are two reasons for this: (1) duplicate tuples produced by the graph generation routine were eliminated, and (2) the locality of a graph provided an upper bound on the actual out-degree of each node (see especially graph $G10$).

6. Performance Study Results

We now present and analyze the results of our performance study. We first discuss queries that compute the complete transitive closure, and we then discuss queries that compute partial transitive closure with varying selectivity. The queries were run (multiple times) on all 12 graphs in Table 2 and the results are presented using specific graphs that exemplify the general trends.

6.1 I/O and CPU Cost Breakdown — General Trends

Table 3 presents a breakdown of the execution cost of algorithm *BTC* for computing the full closure of graph $G6$ with 10, 20 and 50 buffer pages. The experiments were run on a DECstation 5000/25 with 24 MB of memory and an RZ24 SCSI local disk. The real time, user time and system time were obtained using the Unix™ *time* command, while the number of page I/O's was recorded by the simulated buffer manager. The last column in Table 3 shows the estimated total I/O time for each experiment if I/O's were not simulated. This number was calculated by multiplying the number of simulated I/O by 20ms, the approximate time for a single I/O on the machine. (The 20ms value was established by a separate set of experiments and is consistent with the disk specifications.) All times given in Table 3 are in seconds.

Buffer Size	Real Time	User Time	System Time	Simul. I/O's	Est. I/O Time
10	52.50	47.28	4.03	10764	215.28
20	51.58	47.20	2.83	9684	193.68
50	49.35	47.35	0.63	8047	160.94

TABLE 3. I/O and CPU Cost of BTC ($G6$, CTC, M = 10-50)

Comparing the measured user time with the estimated I/O time we see that the closure computation is clearly I/O bound for all three buffer pool sizes used in these experiments. Further analysis of the I/O breakdown between the preprocessing (restructuring) and computation (expansion) phases shows that the computation phase dominates the I/O cost for all buffer sizes. Regarding the CPU cost, we found it to be dominated by the cost of operations on successor lists, and in particular, the cost of list union. Duplicate elimination using bit vectors was found to be quite cheap (see [7] for more details).

6.2 Computing Full Closure

We present the full closure (CTC) results by comparing *BTC* with the other algorithms. For CTC computation, algorithm *BJ* is identical to *BTC*, since it cannot eliminate any non-source nodes. We therefore focus on the *HYB*, *SPN*, *JKB* and *JKB2* algorithms.

Figure 6 compares the I/O cost of *BTC* and *HYB* with respect to the available buffer size M on graph $G6$. The number in the legend of each of the curves for *HYB* is the value of *ILIMIT*, the ratio of the buffer pool reserved for the diagonal block successor lists. When *ILIMIT* is increased, the cost of *HYB* increases as

graph name	out deg. F	generation locality l	number of arcs $\|G\|$	max. node level	graph height H	graph width W	average arc locality	average irredundant locality	closure size $\|TC(G)\|$
$G1$	2	20	3892	297	108	36	34	8	1124406
$G2$	2	200	4053	52	20	202	8	3	674123
$G3$	2	2000	4393	25	11	399	5	2	125610
$G4$	5	20	8605	573	253	34	32	5	1750499
$G5$	5	200	9876	115	55	179	11	5	1497537
$G6$	5	2000	9984	48	29	344	10	5	563333
$G7$	20	20	23365	1192	581	40	21	1	1948375
$G8$	20	200	32724	335	174	214	20	4	1883612
$G9$	20	2000	38731	152	106	365	34	6	1463591
$G10$	50	20	33025	1605	798	41	18	1	1974648
$G11$	50	200	82676	610	317	260	34	3	1948217
$G12$	50	2000	92381	273	188	491	65	6	1778046

TABLE 2. Graph Parameters

well. In fact, the algorithm performs best when no blocking is used, in which case it is identical to *BTC*.

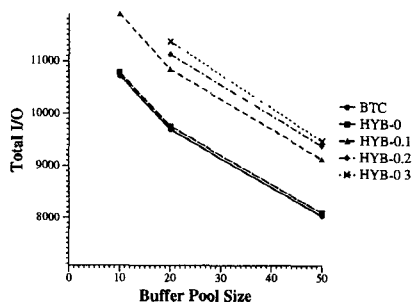

Figure 6. Hybrid vs. BTC, Effect of Blocking, Full Closure ($G9$, M = 10-50)

The main benefit of blocking is that once an off-diagonal successor list is brought into memory, it may be joined with several diagonal block lists. For example, Figure 2 shows an off-diagonal list that may be joined with 3 diagonal block successor lists Blocking was found a useful in reducing I/O in the Direct algorithms presented in [3]. It is therefore surprising that blocking has a detrimental effect on the I/O cost of the Hybrid algorithm. However, there is a major difference between the Direct algorithms and the Hybrid algorithm — the Hybrid algorithm makes use of the immediate successor optimization, which the Direct algorithms do not utilize. In the Direct algorithms, when a successor list is processed, it is joined with it the successor lists of *any* of its successors. In the Hybrid algorithm, when a successor list is processed, it is joined with it the successor lists of its *immediate* successors only. Thus, the number of potential list unions (represented by small squares in Figure 2) is much smaller in the Hybrid algorithm.

At the same time, blocking may degrade the performance of the Hybrid algorithm for several reasons. First, since the diagonal block lists are fixed in memory, the buffer pool becomes effectively smaller. Second, expansion of the diagonal block lists may cause reblocking, i.e., discarding pages holding other successor lists from the buffer pool. Third, in the Hybrid algorithm the off-diagonal part of each diagonal list is processed before the diagonal part. This may cause the algorithm to expand redundant arcs, arcs that would not be expanded if a strict right-to-left order, corresponding to the topological order, was used. With these negative factors combined, the cost of blocking in the Hybrid algorithm becomes greater than its benefit.

Figure 7 (a) compares the I/O cost of *BTC* and the successor tree algorithms *SPN*, *JKB* and *JKB2*, with respect to the average node degree on graphs generated with locality 200 with a buffer pool of 20 pages. The *BTC* algorithm performs better than the other algorithms since its flat successor lists occupy less space then the successor trees of the other algorithms. The successor tree algorithms actually fetch a smaller number of successors (see Section 3.5). However, the reduced "tuple I/O" does not translate into reduced page I/O. When processing an arc (i,j), the first page of the successor tree of j must always be accessed. Hence, in general, the union of two successor trees, S_i and S_j, may save page I/O compared to a successor list union only if two conditions are met: (i) the successor tree of j spans multiple pages, and (ii) there is at least one page of j's successor tree such that all the successors of j on that page need not be added to the successor tree of i (since they are already there, or, in the case of *JKB* and *JKB2*, since they are not special with respect to i). In our experiments we found that almost always each of the pages holding the successor tree of j had to be accessed, and real (vs. tuple) I/O was not saved.

As Figure 7 (a) illustrates, the *SPN* algorithm closes the gap with the *BTC* algorithm as the out-degree goes up, as the relative overhead of storing parent nodes in the successor tree becomes smaller. The *JKB* and *JKB2* algorithms continue to perform poorly since the cost of the restructuring phase (building *predecessor* trees) is higher for these algorithms. The preprocessing cost for *JKB* becomes very high for a high out-degree, while the preprocessing cost for *JKB2* is approximately twice that of *BTC* (see Section 3.6).

The successor tree algorithms generate far fewer duplicates than the flat successor lists algorithms. Figure 7 (b) shows the number of duplicates generated by *BTC* and *SPN* for the same graphs used in Figure 7 (a). However, since duplicate elimination is performed in memory, this cost is not reflected in our model. Further, we believe that the CPU cost of in-memory duplicate elimination can be made negligible, e.g., by using bit vectors, as we did in our implementation. As an example, we analyzed the profile of the computation of the full closure of graph $G6$ using

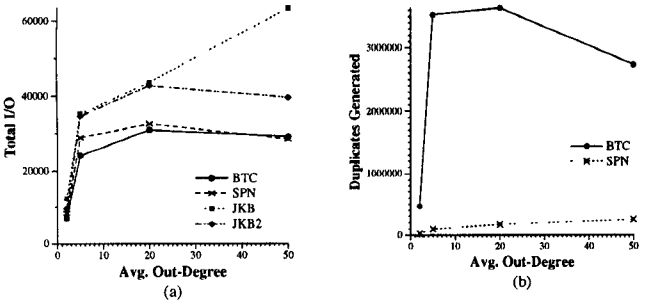

Figure 7. The Successor Tree Algorithms vs. BTC, Full Closure ($M = 20$)

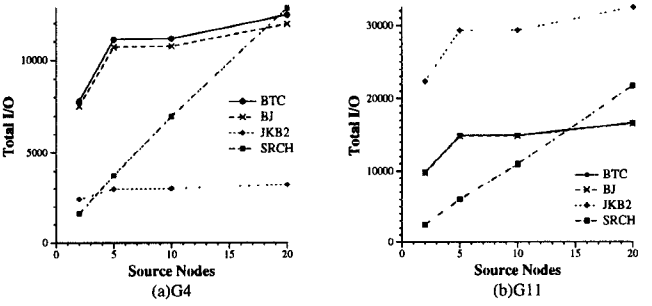

Figure 8. High Selectivity ($G4$ and $G11$, $M = 10$)

algorithm *BTC* with 50 buffer pages. The cost of duplicate elimination using bit vectors was found to be less than 6% of the total CPU cost of computing the closure. Finally, however, we should note that in addition to determining reachability between two nodes in the graph, the successor tree algorithms also establish a path between the two nodes. This additional information, if needed, may justify the higher I/O cost of these algorithms.

The performance of the *HYB* and *SPN* algorithms relative to *BTC* was similar for experiments involving selection. The performance of *JKB* was almost always inferior to that of *JKB2*, and the cost of the preprocessing phase for *JKB* when the out-degree of the graph was high was prohibitly expensive. Therefore, we omit algorithms *HYB*, *SPN* and *JKB* from the discussion of partial transitive closure.

6.3 Computing Partial Closure

The computation of the partial transitive closure displays a high variance in the relative performance of the algorithms over different data sets. Overall, the I/O cost of *BJ* is slightly lower than that of *BTC*. The *SRCH* algorithm performs the best when the number of source nodes is small, but its cost increases rapidly as the number of source nodes is increased. The cost of *JKB2* varies greatly with respect to the other algorithms. We consider first PTC computations with high selectivity (2-20 source nodes), and than with low selectivity (200-2000 source nodes).

6.3.1 High Selectivity

Figures 8 (a) and (b) show two extreme examples of selections with high selectivity. Both graphs display total I/O with respect to s, the number of source nodes, with a buffer pool size of 10 pages. In Figure 8 (a), with $G4$ being the input graph, *JKB2* performs about 1/3 the I/O of algorithms *BTC* and *BJ*. In Figure 8 (b), with $G11$ being the input graph, *JKB2* performs about 2-3 times more I/O than the other algorithms. How do we explain this behavior?

We identify two major factors that determine the I/O cost of the algorithms for computing selections. We describe these factors in the following subsections.

6.3.2 Selection Efficiency

First, we observe that in order to compute the closure of the source nodes in S, the PTC algorithms may compute the closure of other non-source nodes in the magic graph. Let *tc* be the number of tuples generated by an algorithm, and *stc* be the number of those tuples that are part of the expanded successor list of a node in S. We call the ratio $\frac{stc}{tc}$ the *selection efficiency* of the algorithm, and denote it by *se*. The value of *stc* must be the same for all algorithms, since this is the number of tuples in the answer to the query. On the other hand, the value of *tc*, and consequently, the value of *se*, differs between the algorithms. A high value of *se* indicates that the algorithm computes the selection efficiently, in the sense that most of the tuples that it produces are directly part of the answer to the query.

By definition, the *SRCH* algorithm achieves a selection efficiency of 1. In contrast, the *BTC* algorithm exhibits a poor selection efficiency since it completely expands all of the successor lists of nodes in the magic graph. The *BJ* algorithm is somewhat more efficient since it avoids expanding the successor lists of single-parent nodes. Algorithm *JKB2* also expands the successor lists (i.e predecessor trees) of each node in the magic graph. However, these lists are *partial* — only *special* nodes are included in each list, and the number of such nodes is at most $2s$.

Figures 9 (a) and (b) show the values of *tc* for algorithms *BTC*, *BJ*, *SRCH* and *JKB2* for the queries of Figures 8 (a) and (b). Figures 9 (c) and (d) show the selection efficiency exhibited by the algorithms for these queries.

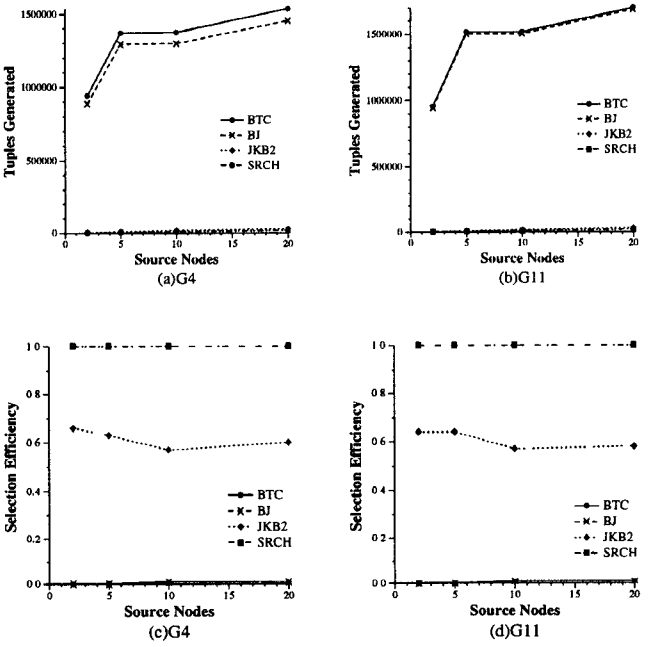

Figure 9. Selection Efficiency ($G4$ and $G11$, $M = 10$)

As expected, the selection efficiency of algorithms *BTC* and *BJ* is poor. Algorithm *BJ* displays a higher selection efficiency than *BTC*. Algorithm *JKB2* achieves 60%-70% of the optimal selection efficiency exhibited by the *SRCH* algorithm, generating less than 1% of the number of tuples produced by *BTC* and *BJ*.

6.3.3 Marking Utilization

Figures 10 (a) and (b) show the number of successor list unions performed by algorithms *BTC*, *BJ*, *SRCH* and *JKB2* for the queries of Figures 8 (a) and (b).

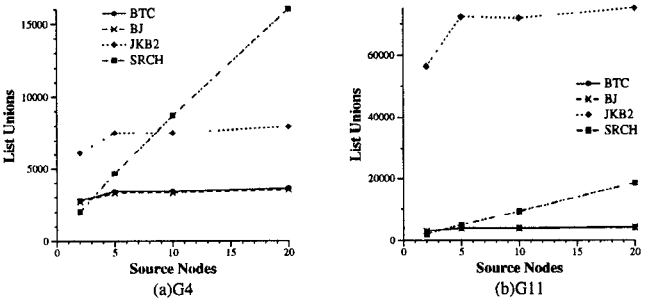

Figure 10. Successor List Unions ($G4$ and $G11$, $M = 10$)

The number of unions performed by the *SRCH* algorithm increases rapidly as the number of source nodes for the query goes up, causing the performance of the algorithm to deteriorate. This is because the the *SRCH* algorithm does not utilize the immediate successor optimization. The number of unions performed by *BTC* and *BJ* is almost identical (*BJ* is slightly lower than *BTC* because it does not expand single-parent nodes). The number of unions performed by *JKB2* is much higher than that of *BTC* and *BJ*.

Algorithm *JKB* performs so many unions because it misses many opportunities to apply the marking optimization. This is because the algorithm uses *partial* successor lists (i.e. predecessor trees) — the list of a node i only records nodes that are special with respect to i. As an example, consider the magic graph of Figure 1 (b), and the predecessor trees for nodes f and j shown in Figures 4 (a) and (c) respectively.

Node d, which is a child of nodes f and j in the inverse graph, does not appear in the tree of f. Therefore, the processing of the arc (j,f) does not result in the marking of the arc (j,d). When the arc (j,d) is later considered, a union of the tree of d with the tree of j is carried out. This union cannot contribute any new arcs to the tree of j; all of the predecessors of d that are special with respect to j (in this case, only node a) have already been inserted to the tree of j as a result of processing the arc (j,f). This redundant union requires the predecessor tree of d to be in memory, and may cause an I/O if it is not already there.

Figures 11 (a) and (b) show the percentage of arcs that were marked by algorithms *BTC*, *BJ* and *JKB2* for the queries of Figures 8 (a) and (b) (this percentage is 0 for algorithm *SRCH*). The marking percentage is almost 0 for *JKB2*, showing that the algorithm misses almost all of the opportunities to apply the marking optimization.

Another consequence of the missed marking opportunities is that the unions performed by *JKB* are more "expensive" than the unions performed by the other algorithms. Figures 12 (a) and (b)

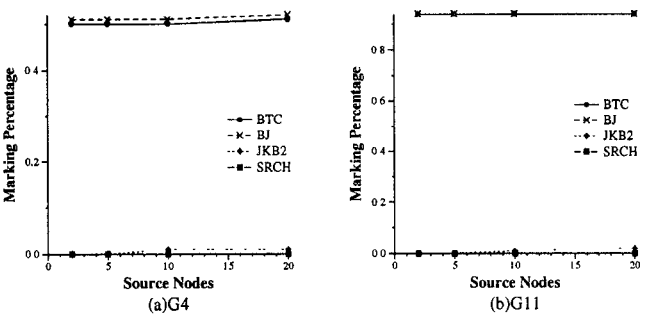

Figure 11. Marking Percentage ($G4$ and $G11$, $M = 10$)

show the average locality of unmarked arcs for the four algorithms for the queries of Figures 8 (a) and (b). The locality is much worse for *JKB*. As we discussed in Section 5.3, the worse this locality, the higher the probability that a child's successor list is not found in the buffer pool and that an I/O will thus be required.

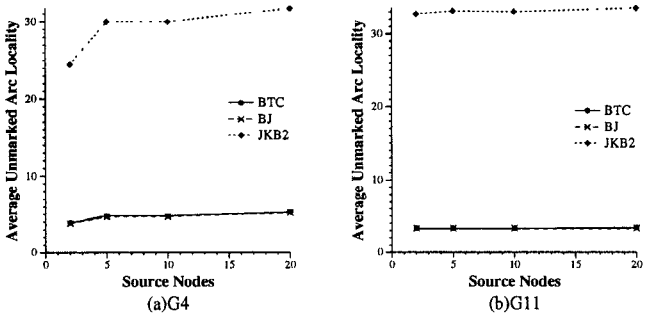

Figure 12. Avg. Irredundant Arc Locality ($G4$ and $G11$, $M = 10$)

We observe that the poor marking utilization problem may arise with any algorithm that uses incomplete successor lists (or trees), independent of the specific criterion by which the nodes in the lists are chosen (i.e., the definition of *special* nodes).

6.3.4 Comparing *JKB2* to *BTC*

When comparing the *SRCH* algorithm with *BTC* it is clear that the selectivity of the query is the dominant factor affecting the performance of *SRCH*. As expected, the performance of *SRCH* deteriorates as the number of source nodes specified in the query increases. A comparison of *JKB2* and *BTC* is more difficult, since the two factors displayed in the last two subsections, *selection efficiency* and *marking utilization*, combine to determine the relative performance of these algorithms. Given the great differences between the algorithms along these dimensions, the following question arises — can we suggest criteria by which to decide which of these algorithms will do better for the computation of a PTC query on a given graph?

We suggest that the I/O cost of *JKB* relative to *BTC* depends on the *width* of the input graph (see Section 5.3). Table 4 lists the width of graphs $G1$ through $G12$ and the Total I/O cost of *JKB* relative to *BTC* for computing PTC with 5 and 10 source nodes with a buffer pool pf 10 pages. The graphs are sorted in increasing width order, and the I/O cost of *JKB* is normalized with respect to the cost of *BTC* for the same queries.

Generally speaking, algorithm *JKB* performs well when the width of the graph is low, but badly when the width of the graph

Graph	$G4$	$G1$	$G7$	$G10$	$G5$	$G2$	$G8$	$G11$	$G6$	$G9$	$G3$	$G12$
Width	34	36	40	41	179	202	214	260	344	365	399	491
$s = 5$	0.27	0.39	0.43	0.6	0.35	0.86	0.76	1.97	1.1	1.92	1.54	3.24
$s = 10$	0.28	0.38	0.43	0.6	0.39	0.9	0.80	1.97	1.32	1.86	1.42	3.21
Height	253	108	581	798	55	20	174	317	29	106	11	188

TABLE 4. Comparing *JKB* and *BTC* for PTC Queries

is high. This is because of the poor marking utilization of *JKB* which, on wide graphs, causes an excessive number of list unions. The size of the lists being unioned is of secondary importance compared with the number of these unions. Consequently, *JKB* is sensitive mostly to the width of the graph, and is less sensitive to its height. (We have not observed a similar correlation for the *SRCH* algorithm relative to *BTC* or *JKB*2.) To illustrate this observation we also list the height of graphs $G1$ through $G12$ on the bottom of Table 4.

6.3.5 Effect of Buffer Pool Size

Figures 13 (a) and (b) show the total I/O cost of algorithms *BTC*, *JKB*2 and *SRCH* for graphs $G4$ and $G11$ for a PTC computation with 5 source nodes as the buffer pool size is increased from 10 to 50. Figures 13 (c) and (d) show the buffer pool hit ratio of the algorithms for the queries and buffer sizes of Figures 13 (a) and (b). For *BTC* and *JKB*2 the hit ratio does not take into account the preprocessing phase, i.e., it gives the percentage of successor list page requests during the computation phase that were satisfied from the buffer pool.

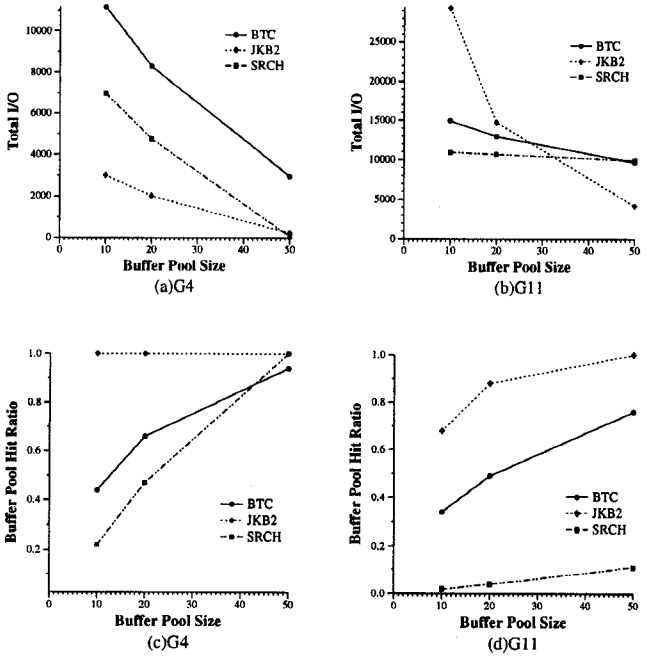

Figure 13. High Selectivity — Effect of Buffer Pool Size ($G4$ and $G11$, 10 Source Nodes)

The performance of all three algorithms improves as the buffer pool size goes up since the buffer pool hit ratio improves. The *JKB*2 algorithm is the most sensitive to the buffer pool size. With a buffer page size of 10 for graph $G4$ and of 50 for graph $G11$ the computation of the closure by algorithm *JKB*2 becomes memory resident and the algorithm performs almost no I/O's during the computation phase. This is because the successor lists (i.e. predecessor trees) used by *JKB*2 are much smaller on average than the successor lists used by *BTC*, since they contain only special nodes (see Section 3.6). The I/O cost of *JKB*2 at this point is dominated by the cost of the preprocessing phase. The reader should keep in mind, however, that *JKB*2 is an implementation of the Compute_Tree algorithm that relies on a dual representation of the input graph by two relations with associated clustered indices (see Section 4.1). Without such a dual representation the cost of the preprocessing phase for the Compute_Tree algorithm can be much higher (for example, observe the different curves for *JKB* and *JKB*2 in Figure 7 (a)).

6.3.6 Low Selectivity

For the low selectivity PTC queries (200-2000 source nodes) the cost of the *SRCH* algorithm is much higher than the other algorithms, usually by 1-2 orders of magnitude. This is to be expected since the algorithm performs an independent search from each of the source nodes. The other algorithms exhibit similar trends to the high selectivity queries, but the magnitudes of the differences between them are much smaller. Figures 14 (a) - (d) show respectively the total I/O, number of tuples generated, marking percentage, and number of list unions for *BTC*, *BJ* and *JKB*2 for graph $G9$ with 20 buffer pages.

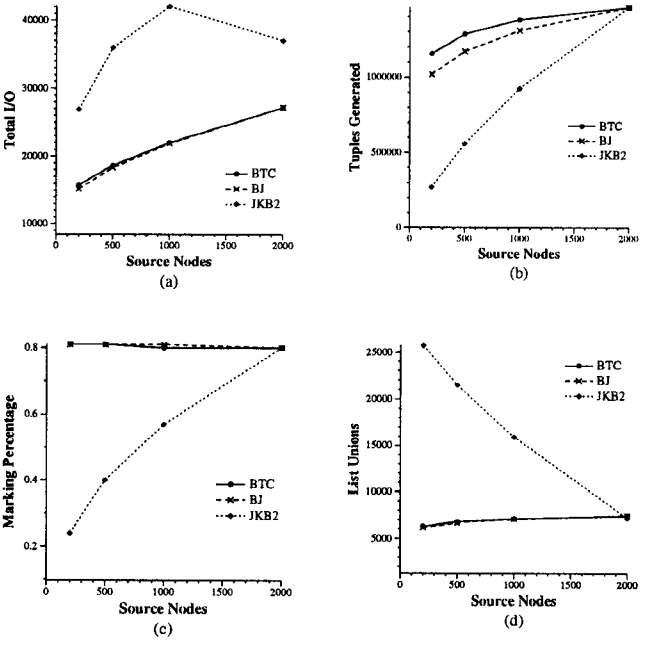

Figure 14. Low Selectivity Trends ($G9$, 20 buffer pages)

Algorithm *BJ* performs almost the same as *BTC* in this range, since it can find few non-source single-parent nodes to eliminate. Considering algorithm *JKB*2, both the positive aspects of the

algorithm's use of partial successor lists (higher selection efficiency, less tuples generated) and the negative aspects (lower marking percentage, more unions) diminish as s approaches 2000. This is because the number of special nodes in the successor trees created by *JKB2* increases with s. Finally, at s = 2000 (full closure), the three curves converge. The total I/O curve is higher for *JKB2* even at this point because of the extra parent information stored in its successor trees (see Section 3.5). The performance of *JKB2* is actually worse at s = 1000 than at s = 2000, which we attribute to the fact that at s = 1000 the algorithm still misses about 30% of the marking opportunities, consequently performing many redundant unions.

7. Evaluation Methodology

A large number of different cost models have been used in studies of algorithms for transitive closure and recursive query evaluation, making a comparison of such algorithms based on a literature survey quite difficult. The cost metrics used in the literature include *main memory operations* [8, 9, 22-24, 26, 27], *number of deductions* (i.e. number of tuples generated including duplicates) [5, 14, 15, 21], *number of distinct tuples derived* [20], *I/O complexity* [25], *successor list I/O* [13, 17, 18], *number of successor list unions* [17, 18], *tuple I/O* [2, 6], *page I/O* [3, 12, 19], and finally *I/O + CPU time* [3, 12, 19]. (The total elapsed time may be measured directly [3], but it is sensitive to the workload of the system.)

A significant consequence of our study is a better understanding of the impact that the choice of cost metrics can have on the results of a performance evaluation. Since the computation of the transitive closure of a large graph is likely to be I/O bound (see Section 6.1 and [3, 12, 19]), an important question to be asked is — can cost metrics which count operations at the tuple or successor list level be used to estimate the I/O cost of a transitive closure computation?

We believe that the results of Section 6 clearly demonstrate that such extrapolation is highly unreliable. We substantiate this claim with a few examples.

- Based on the number of tuples produced, or the tuple I/O (or I/O complexity) metric, the successor spanning tree algorithms would appear to perform much better than the *BTC* algorithm for the computation of the full closure (Figure 7 (a)), while in fact they perform worse.
- Based on the number of distinct tuples derived, algorithm *JKB2* would appear to always outperform algorithm *BTC* for the computation of partial closure with high selectivity (Figures 9 (a) and (b)). On the other hand, based on the number of successor list unions, or the successor list I/O metric, the opposite conclusion would be drawn (Figures 10 (a) and (b)). As we demonstrate, neither of these conclusions holds in all cases (Figures 8 (a) and (b)).

Our conclusion is that a reliable evaluation of the page I/O cost of a transitive closure computation can only be obtained via a performance study that directly considers that I/O cost, rather than trying to estimate it using higher level metrics and postulating some correlation between these higher level metrics and the page I/O metric.

8. Related Work

Many algorithms have been suggested in recent years for the computation of the full or partial transitive closure of a large graph. However, most previous performance investigations have involved a small set of algorithms and data sets, and only in a few studies were the algorithms actually implemented and the I/O cost directly measured.

Agrawal et al. [1, 3] studied the Blocked Warshall (aka Blocked Column) and Blocked Warren (aka. Blocked Row) Direct algorithms and the seminaive algorithm. They measured page I/O and elapsed time and found I/O to be the dominant cost factor. The Direct algorithms were found to perform much better for the computation of CTC over all input graphs. Kabler et al. [19] explored several implementations of the Seminaive, Smart and Blocked Warren algorithms using different join methods, duplicate elimination techniques, and buffer management policies. They measured page I/O and elapsed time. They found Seminaive to always outperform Smart. Blocked Warren was overall the best algorithm for CTC, but for PTC computation with less than 1/3 of the nodes being source nodes Seminaive performed better. The results of [1, 3, 19] demonstrated that the matrix-based algorithms out-perform the iterative algorithms for the computation of the full transitive closure by a wide margin. However matrix-based algorithms were not efficient when computing partial closure with high selectivity (small number of source nodes).

Ioannidis et al. [12] studied the performance of several graph-based algorithms including Schmitz [23], BTC, and the more complex DFS algorithms presented in [12]. They measured page I/O and CPU time. Overall, BTC was found to be best with regard to both I/O and CPU cost. Ioannidis et al. found that the algorithms were usually I/O bound. They also compared the performance of the graph based algorithms to the performance of the Seminaive, Smart and Blocked Warren algorithms, using the results of [19], for similar graphs and buffer sizes. The graph-based algorithms were superior over all input graphs, with an order of magnitude difference for cyclic graphs.

Other performance studies, which did not measure page I/O, should be mentioned. Agrawal and Jagadish measured tuple I/O in a study that included the Hybrid and Blocked Warren algorithms, the Grid algorithm and a DFS algorithm [2]. Dar and Jagadish measured tuple I/O while comparing the Spanning Tree and Hybrid algorithms. Jiang measured successor list I/O and unions of in an investigation of several graph-based algorithms for PTC [17, 18]. Taken together, the results reported in [2, 6, 12, 17] clearly demonstrate that the graph-based algorithms and their hybrid variants are superior to the iterative and matrix-based algorithms, both for the computation of both CTC and PTC. We therefore focused our performance study on graph-based and hybrid algorithms, selecting the algorithms that were the best performers in the earlier studies. As suggested in [15, 17], for PTC with high selectivity, a simple search from the source nodes may be the most efficient strategy. Hence, we also added the Search algorithm to our suite.

9. Conclusions

We have studied the performance of reachability algorithms for computing the full and partial transitive closure. Our investigation was based on a detailed simulation of the algorithms using a uniform implementation framework. The following are the main conclusions of our performance study:

1. The Hybrid algorithm and the Spanning Tree algorithms generally performed worse than the other algorithms. In the case of Hybrid, we attributed that to the ineffectiveness of the blocking technique used by the algorithm. In the case of the Spanning Tree algorithms, we have shown that the savings in tuple reads and in the number of duplicates produced does not necessarily translate into an improved I/O behavior.
2. Jiang's single-parent optimization offered a (small) performance improvement for PTC queries, especially for high selectivity queries over graphs with a small out-degree.
3. The performance of Jakobsson's algorithm varied widely for high selectivity queries. We identified two factors, selection efficiency and marking utilization, that determine the performance of the algorithm for such queries. We also developed a novel model for characterizing directed acyclic graphs, and we presented some evidence that this model may be useful in understanding the performance of Jakobsson's algorithm relative to the BTC algorithm for PTC queries.
4. The Search algorithm performed best for high selectivity queries. As anticipated, its performance deteriorated rapidly as the number of source nodes was increased.
5. In addition to directly comparing specific transitive closure algorithms, we investigated the evaluation methodology for such algorithms. Our results indicate that performance metrics based on tuples or successor lists, used in many previous studies, simply cannot be used reliably to evaluate the I/O cost of transitive closure computation.

REFERENCES

[1] R. Agrawal and H. V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations", *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 255-266.

[2] R. Agrawal and H. V. Jagadish, "Hybrid Transitive Closure Algorithms", *Proc. 16th Int'l Conf. Very Large Data Bases*, Brisbane, Australia, 1990.

[3] R. Agrawal, S. Dar and H. V. Jagadish, "Direct Transitive Closure Algorithms: Design and Performance Evaluation", *ACM Trans. Database Syst. 15*, 3 (Sep. 1990), .

[4] A. V. Aho, E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1975.

[5] C. Beeri and R. Ramakrishnan, "On the Power of Magic", *Proc. 6th Symp. Principles of Database Systems*, San Diego, California, March 1987, 269-283.

[6] S. Dar and H. V. Jagadish, "A Spanning Tree Transitive Closure Algorithm", *IEEE 8th Int'l Conf. on Data Engineering*, Phoenix, Arizona, Feb. 1992, 2-11.

[7] S. Dar, "Augmenting Database with Generalized Transitive Closure", Univ. Wisconsin, Madison, Wisconsin, 1993. Ph.D. Dissertation.

[8] J. Ebert, "A Sensitive Transitive Closure Algorithm", *Information Processing Letters 12*, (1981), 255-258.

[9] J. Eve and R. Kurki-Suonio, "On Computing the Transitive Closure of a Relation", *Acta Informatica 8*, (1977), 303-314.

[10] A. Goralcikova and V. Koubek, "A Reduct and Closure Algorithm for Graphs.", *Proceedings of the Int'l Conf. on Mathematical Foundations of Computer Science*, 1979, 301-307.

[11] K. A. Hua and J. X. W. Su, "Efficient Evaluation of Traversal Recursion Using Connectivity Index", *Proceedings 9th Int'l Conf. on Data Engineering*, Vienna, Austria, April 1993.

[12] Y. E. Ioannidis, R. Ramakrishnan and L. Winger, "Transitive Closure Algorithms Based on Depth-First Search", *ACM Trans. Database Syst.*, . to appear.

[13] Y. E. Ioannidis and R. Ramakrishnan, "An Efficient Transitive Closure Algorithm", *Proc. 14th Int'l Conf. Very Large Data Bases*, Los Angeles, California, Aug.-Sept. 1988.

[14] H. Jakobsson, "Mixed-Approach Algorithms for Transitive Closure", *Proc. 10th Symp. Principles of Database Systems*, Denver, Colorado, 1991, 199-205.

[15] H. Jakobsson, "On Tree-Based Techniques for Query Evaluation", *Proc. 11th Symp. Principles of Database Systems*, 1992, 380-392.

[16] H. Jakobsson, "Mixed-Approach Algorithms for Transitive Closure", Unpublished Manuscript, Computer Science Dept., Stanford Univ., Stanford, California, 1993.

[17] B. Jiang, "Design, Analysis, and Evaluation of Algorithms for Computing Partial Transitive in Databases", Computer Science Tech. Rep., ETH Zurich, June 1990.

[18] B. Jiang, "A Suitable Algorithm for Computing Partial Transitive Closures in Databases", *Proc. IEEE 6th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1990.

[19] R. Kabler, Y. E. Ioannidis and M. Carey, "Performance Evaluation of Algorithms for Transitive Closure", *Information Systems 17*, 5 (Sep. 1992), .

[20] I. Mumick and H. Pirahesh, "Overbound and Right-Linear Queries", *Proc. 10th Symp. Principles of Database Systems*, Denver, Colorado, May 1991, 127-141.

[21] J. F. Naughton, R. Ramakrishnan, Y. Sagiv and J. D. Ullman, "Efficient Evaluation of Right-, Left-, and Multi-Linear Rules", *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 235-242.

[22] P. Purdom, "A Transitive Closure Algorithm", *BIT 10*, (1970), 76-94.

[23] L. Schmitz, "An Improved Transitive Closure Algorithm", *Computing 30*, (1983), 359-371.

[24] C. P. Schnorr, "An Algorithm for Transitive Closure with Linear Expected Time", *SIAM J. Computing 7*, 2 (May 1978), 127-133.

[25] J. D. Ullman and M. Yannakakis, "The Input/Output Complexity of Transitive Closure", *Proc ACM-SIGMOD 1990 Int'l Conf. on Management of Data*, Atlantic City.

[26] H. S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations", *Commun. ACM 18*, 4 (April 1975), 218-220.

[27] S. Warshall, "A Theorem on Boolean Matrices", *J. ACM 9*, 1 (Jan. 1962), 11-12.

[28] M. Yannakakis, "Graph-Theoretic Methods in Database Theory", *Proc. 9th Symp. Principles of Database Systems*, Nashville, Tennessee, April 1990, 230-242.