Information Systems Vol. 17, No. 5, pp. 415–441, 1992
Printed in Great Britain.

0306-4379/92 $5.00 + 0.00

PERFORMANCE EVALUATION OF ALGORITHMS FOR TRANSITIVE CLOSURE

ROBERT KABLER,† YANNIS E. IOANNIDIS‡ and MICHAEL J. CAREY
Computer Sciences Department, University of Wisconsin, Madison, WI 53706, U.S.A.

(Received 5 July 1991; in revised form 28 February 1992)

Abstract—This paper presents the results of an experimental evaluation of the performance of three main algorithms for transitive closure: Seminaive, Smart and Blocked Warren. The algorithms have been implemented using a variety of join methods (block nested-loops and hash-join), disk-based and memory-based data structures and buffer replacement strategies. The algorithms were tested on several graphs, ranging from regular trees to random acyclic graphs to random general graphs. Contrary to what several previous studies have found, our experiments indicate that Seminaive is almost always superior to Smart. In most cases, Seminaive exhibited inferior performance to Warren, but surprisingly, there are some types of graphs where Blocked Warren generates more duplicates than Seminaive and is therefore slower. Finally, for the common case where a transitive closure query involves a selection, Seminaive can take advantage of the constants in the selection, whereas Blocked Warren and Smart cannot. Our experiments indicate that the percentage of the graph nodes that need to be selected for Blocked Warren to be superior to Seminaive is rather large (for all graphs tested, it must be greater than 1/3). This implies that for the majority of transitive closure queries with selection, Seminaive is the preferred strategy.

Key words: Transitive closure, recursion, node reachability

1. INTRODUCTION

Graphs offer a very useful data abstraction that captures binary relationships defined on a set of elements. They are very widely used to represent hierarchies (e.g. part hierarchies, genealogical trees or management hierarchies) and networks (e.g. computer networks, production lines or semantic networks). Many important questions that one may want to ask of data organized in a graph essentially require the computation of the *transitive closure* of the graph. For example, consider an airline reservation system that uses a graph to capture the network of cities where all airlines fly, together with the prices of each flight. Questions of interest include "the cities reachable from Madison on United", "the cheapest fair on flights between San Francisco and Munich" or "the pairs of cities that are connected with flights of the same airline". Answering any of the above queries requires computing (part of) the transitive closure of the graph (network) of flights. Conventional relational query processing technology cannot perform such computations without resorting to the general processing capabilities of a programming language, since transitive closure is not expressible in relational algebra [1]. To overcome this, several earlier systems have extended their query language with special transitive closure constructs [2], while many systems currently under development support linear recursion, which captures transitive closure as a special case. This paper describes the implementation and performance evaluation of several transitive closure algorithms for disk-resident graphs.

The problem of computing the transitive closure of a directed graph in a disk-based environment has received considerable attention in the past few years. Several papers have appeared that propose new algorithms [3–5], discuss disk-based implementation techniques for old algorithms and study their performance [6–9], investigate parallel evaluation techniques [10, 11], and study theoretical optima and bounds [12]. Despite these many references, several questions regarding transitive closure evaluation remain unanswered. This paper belongs to the second category mentioned above, i.e. it is an investigation of alternative implementations of known algorithms and an evaluation of their performance. The results that it presents complement those of previous studies

†Present address: Epic Systems Corp., 5609 Medical Circle, Madison, WI 53719, U.S.A.
‡To whom correspondence should be addressed.

and offer new insights on how the available algorithms should be implemented for improved performance. The contributions of the paper are in the following directions: (a) join method, data structure and buffer management alternatives for the implementation of Seminaive and Smart, (b) effect of data characteristics on the behavior of algorithms and (c) algorithm performance in the presence of selections. We motivate each one of these directions below.

(*a*) *Implementation alternatives*: The importance of studying fixpoint evaluation algorithms, i.e. Seminaive and Smart, when used for the specialized problem of transitive closure computation lies in their generality. Although specialized algorithms like Blocked Warren have been shown to perform better than Seminaive [6], they are not applicable to more general forms of recursion. Studying the implementation and performance characteristics of the above two algorithms on the specific problem of transitive closure should offer useful knowledge about their behavior in the general setting. This paper investigates several implementation alternatives for them, studies their performance trade-offs, and concludes with suggestions regarding which one should be adopted and how it should be implemented. Several of the previous studies have adopted a single implementation of each algorithm, which was not necessarily the most advantageous one, while others have focused on introducing novel techniques and studying their effect without trying to gain a more global perspective.

(*b*) *Data characteristics*: The complexity of the transitive closure operation makes it difficult to identify the data characteristics that affect the performance of alternative algorithms or the precise nature of their effects. To the best of our knowledge, no previous study has studied this issue in any detail. Although some studies have supported their conclusions by extensive experiments that involved diverse sets of graphs, efforts to make the relationship between data characteristics and algorithm performance explicit have been incomplete. This paper moves in the direction of filling this gap and provides evidence that no algorithm is universally superior, i.e. for any particular algorithm there are graphs on which the algorithm is the most efficient.

(*c*) *Selections*: It is expected that most often the transitive closure operation will be performed in combination with a selection. This introduces a new dimension in comparing algorithms, as some of them cannot take advantage of selections whereas others can. As expected, in the presence of very selective operations, the latter perform better than the former. On the other hand, the former usually perform better than the latter when no selection is present. There has been no effort to compare the algorithms over the whole range from highly selective to non-selective queries. This paper presents the results of such an investigation and identifies cross-over points for the performance of the two types of algorithms.

The rest of this paper is organized as follows. Section 2 provides some terminology and describes three algorithms that were used in the experiments. Section 3 contains a description of the implementation of the algorithms, together with our decisions regarding the available alternatives for various aspects of each one. Section 4 describes the testbed used for our experiments, defining the parameters of the algorithms and the graph characteristics on which we focused. In Sections 5, 6 and 7, we discuss the performance of the three algorithms of interest in computing the full transitive closure of trees, the full transitive closure of non-tree graphs, and a selected part of the transitive closure of graphs, respectively. In each case, we present the general trends and also analyze the effects of the interesting parameters on performance. Section 8 contains a comparison of our results to those of related studies. Finally, Section 9 provides some conclusions and directions for future work.

2. TERMINOLOGY AND ALGORITHMS

2.1. Terminology

We denote the initial binary relation by **R**, and its transitive closure by **T**. The first attribute of a binary relation is called the *source* and the second one is called the *destination*. If an arc (A, B) appears in **R**, then B is a *successor* of A. If an arc (A, B) appears in **T**, then B is a *descendent* of A.

The symbol $\circ$ indicates the composition of two binary relations, i.e. an equality join involving the second attribute of the first relation and the first attribute of the second relation followed by

a projection on the non-join columns. For example, given the binary relations **X** and **Y** with columns (a, b) and (b, c), respectively, then

$$\mathbf{X} \circ \mathbf{Y} = \pi_{a,c}(\mathbf{X} \underset{b=b}{\bowtie} \mathbf{Y}).$$

Since it often helps to think of a binary relation as a directed graph rather than as a collection of tuples, we frequently refer to relations in terms of a directed graph rather than as tables of values. The source and destination values comprise the set of nodes in the graph, and the tuples (A, B) correspond to the arcs. Let S be a set of paths in the graph such that, for every pair of nodes in the graph, S contains the shortest path between them. That is, $S = \{p | A$ and B are nodes in the graph and p is the shortest path between A and $B\}$. The length of the longest path in S is the *depth* of the graph.

We use $\mathbf{R}^2$ to denote $\mathbf{R} \circ \mathbf{R}$, i.e. the result of composing **R** with itself. That is, $\mathbf{R}^2$ contains arcs that correspond to paths of length two in the original graph. The above can be generalized to arbitrary powers of **R**. The transitive closure **T** of the binary relation **R** is equal to

$$\mathbf{T} = \bigcup_{i=1}^{\infty} \mathbf{R}^i.$$

That is, **T** contains precisely the arcs (A, B) such that a path exists from node A to node B in the original relation **R**. If during the computation of **T** an arc (tuple) is generated more than once, the second time this happens, it is called a *duplicate arc* (*tuple*).

2.2. *Algorithms*

In this study, we have focused our attention on three algorithms: Seminaive, Smart and Blocked Warren. We briefly describe them in the next subsections.

2.2.1. Seminaive. The Seminaive algorithm has been formally introduced by Bancilhon [13] as an algorithm to compute the fixpoint of any recursive Horn clause, not just transitive closure. The algorithm had been used even before its formal specification [14, 15], and since then it has been studied by several other researchers as well [3, 5, 7, 8]. Pseudocode for Seminaive appears in Fig. 1. The algorithm proceeds in iterations. It uses a relation $\Delta\mathbf{T}$ to contain the new tuples that are produced in each iteration (new in the sense that they have not been produced in any previous iteration). In graph terms, for all i, at the beginning of the ith iteration, $\Delta\mathbf{T}$ contains an arc (A, B) if the shortest path between A and B in **R** has length i. The algorithm stops when $\Delta\mathbf{T}$ becomes empty. Clearly, if d is the depth of the graph that corresponds to **R**, Seminaive requires d iterations to terminate.

Note that, when the graph is acyclic, duplicate elimination is important only for efficiency, not for termination. Termination is guaranteed even without applying the difference operator in line 4 of Fig. 1. Even efficiency alone is enough to render duplicate elimination mandatory in most cases, however. This has been discussed in the past [6] and will be confirmed in Section 6 as well. In contrast, when the graph has cycles, duplicate elimination is important for both efficiency and termination.

Seminaive is easily adapted to a selection query when the selection is on the source, i.e. when the *descendents* of some specific nodes are requested. One only needs to apply the source selection on the tuples that initially populate $\Delta\mathbf{T}$ and **T**. Otherwise, Seminaive remains unchanged and correctly computes the selected transitive closure. If the selection is on the destination, i.e. when the *ancestors* of some specific nodes are requested, one can still use the above algorithm, but the composition in line 4 of Fig. 1 must be modified to $\mathbf{R} \circ \Delta\mathbf{T}$.

```
1.        ΔT = R
2.        T = R
3.        while (ΔT ≠ ∅) {
4.              ΔT = (ΔT ∘ R) − T
5.              T = T ∪ ΔT }
```

Fig. 1. The Seminaive algorithm.

```
1.    ΔT = R
2.    T = R
3.    while (ΔT ≠ ∅){
4.          ΔT = (ΔT ∘ ΔT) − T
5.          T = T ∪ ΔT ∪ (ΔT ∘ T)}
```

Fig. 2. The Smart algorithm.

2.2.2. Smart. The Smart algorithm has been independently proposed by Ioannidis [3] and by Valduriez and Boral [5]. Like Seminaive, it is an algorithm for the computation of the fixpoint of any recursive Horn clause, and it has been studied by several researchers [3, 5–9]. Pseudocode for Smart appears in Fig. 2. The algorithm proceeds again in iterations. Unlike Seminaive, however, **ΔT** participates in two compositions in each iteration, one with itself and one with **T**, the accumulated result at that point. Moreover, for all i, at the beginning of the ith iteration, **ΔT** contains an arc (A, B) if the shortest path between A and B in **R** has length 2^{i-1}, and **T** contains an arc (A, B) if the shortest path between A and B in **R** has length less than 2^i. If d is the depth of **R**, then Smart requires $\lceil \log_2(d+1) \rceil$ interations to terminate. Although Smart requires fewer iterations than Seminaive, this is done at the price of having more expensive join operations and producing more duplicates at each iteration. Details about this trade-off are studied in Sections 5 and 6.

Unlike Seminaive, Smart cannot take full advantage of selections. The best that can be done is to fully compute $\mathbf{R}^{2^i}$ for all necessary powers, i.e. to compute **ΔT** for all iterations until termination, and then make use of the selection as in Seminaive. Although this approach is more efficient than computing the complete transitive closure, it is still likely to be less efficient than Seminaive.

2.2.3. Blocked Warren. The Warren algorithm has been proposed by Warren [16] as a modification to Warshall's algorithm [17] for the computation of the transitive closure of a directed graph represented as an adjacency matrix. Warren's algorithm requires two passes through the adjacency matrix, instead of the single pass of Warshall's algorithm, but has been proved to be more efficient in general.

Agrawal and Jagadish [6] have proposed the Blocked Warren algorithm, which is a further modification of Warren's algorithm that uses lists of descendent nodes instead of bit vectors and

```
/*First Pass*/
for each row partition
    for j = 1 to i_b − 1                    /*Process off-diagonal block in column-order*/
        for i = i_b to i_e
            if (i,j) exists
                Add descendent list of j to that of i
    for j = i_b to i_e                      /*Process lower triangle of diagonal block in column-order*/
        for i = j + 1 to i_e
            if (i,j) exists
                Add descendent list of j to that of i

/*Second Pass*/
for each row partition
    for j = i_b to i_e                      /*Process upper triangle of diagonal block in column-order*/
        for i = i_b to j − 1
            if (i,j) exists
                Add descendent list of j to that of i
    for j = i_e + 1 to n                    /*Process off-diagonal block in column-order*/
        for i = i_b to i_e
            if (i,j) exists
                Add descendent list of j to that of i
```

Fig. 3. The Blocked Warren algorithm.

reduces disk accesses for disk-resident binary relations. It achieves this by processing a collection of nodes together, as a block, rather than one node at a time. These groups of nodes are called *diagonal blocks* and each node is part of exactly one diagonal block during each pass of the processing. Specifically, several consecutive nodes in the matrix form a *row partition*, which includes a diagonal block and two off-diagonal blocks on its two sides. Each one of these blocks of a row partition is processed separately. Pseudocode for Blocked Warren [6] is shown in Fig. 3. The algorithm processes the nodes in a specific order based on their location in the adjacency matrix of the graph. The specific order restrictions are given elsewhere [6]. To a first approximation, the elements to the left of the diagonal are processed in the first pass and the elements to the right of the diagonal are processed in the second pass.

The exact bounds of a diagonal block are established based upon the amount of memory available and the number of descendents of the nodes. In Fig. 3, they are denoted by the subscripts b and e, which represent the first and last node of the diagonal block, respectively. Each pass in Fig. 3 contains two separate loops through the diagonal block. One loop processes the diagonal block nodes with nodes not belonging to the current diagonal block and the other processes them with nodes that do belong to the current diagonal block. There is no restriction that the diagonal blocks must remain the same from one pass to the next—partitioning is done dynamically. Usually, the second pass needs more partitions because the descendent lists of the nodes that need to simultaneously be in main memory are longer than in the first pass. A detailed discussion of the algorithm can be found elsewhere [6].

Unfortunately, Blocked Warren is not efficient with selection queries. It requires the computation of the complete transitive closure of a graph before a selection can be done. This is a disadvantage relative to Seminaive, and even relative to Smart to some extent.

3. IMPLEMENTATION OF ALGORITHMS

As part of this study, these three algorithms have all been implemented, some in multiple versions. This section describes the various implementation alternatives that were considered and the specific choices that were adopted. Due to their similarities, we describe the implementations of Seminaive and Smart together and describe Blocked Warren separately.

3.1. Seminaive and Smart

For all versions of Seminaive and Smart, we assume that each relation is already partitioned into several smaller disjoint ones. This partitioning can be done by any of several methods. In our study, we use a simple arithmetic hash function on one of the attributes of the relation (depending on the implementation) to form the partitions, i.e. each hash bucket is one partition. We use $\mathbf{R}_i$, $\mathbf{T}_i$, and $\Delta\mathbf{T}_i$ to denote the ith partition of **R**, **T**, and $\Delta\mathbf{T}$ respectively. We assume that each file page contains tuples belonging to a single partition. With a large number of partitions this can result in many partially-filled pages; however, this constraint also minimizes the number of pages that must be accessed to read only one partition. In general, in each iteration of Seminaive and Smart, each partition of $\Delta\mathbf{T}$ must be composed with all partitions of **R** (Seminaive) or **T** and $\Delta\mathbf{T}$ (Smart). Depending on the join method and the exact partitioning, however, it may be possible to avoid composing several such partition pairs, by knowing that they produce no results, thus improving performance. A further enhancement in performance stems from the fact that at any instant only a single partition needs to be examined for duplicates rather than the entire relation.

The following is a list of issues on which a decision had to be made for every algorithm: join method, timing of processing, number of files, duplicate elimination, page structure and main memory management. Each of these issues is discussed in one of the subsections that follow.

3.1.1. Join method. We have experimented with two join methods: hash-join and (essentially) block nested-loops†. For hash-join, we used an algorithm similar to the second pass of the "GRACE" algorithm [18], loading a full partition of **R** or **T** in memory and then scanning the corresponding partition of $\Delta\mathbf{T}$ one page at a time. For nested-loops, $\Delta\mathbf{T}$ was the outer relation and

†In ordinary block nested-loops, tuples are visited in the order of their placement in the relations to be joined. In our implementation, they are visited in partition order, i.e. tuples in partition i are visited before those in partition $i + 1$.

R or **T** was the inner one. The decision to use $\Delta\mathbf{T}$ as the outer relation was based on the combination of the following advantages. First, when immediate processing is done (Section 3.1.2) on trees, the newly produced tuples belong to the partition of $\Delta\mathbf{T}$ from which they were generated, which is already in main memory. Thus, they are immediately available for further processing without any I/O. Second, for Seminaive, new tuples can go through the process of duplicate elimination against $\Delta\mathbf{T}$ without any I/O since the appropriate partition is in main memory. Thus, although duplicate elimination against the rest of **T** still involves I/O, some fraction of the cost is avoided. There would be no corresponding gain if **R** were the outer relation in Seminaive, as duplicate elimination is done against **T** (of which part is $\Delta\mathbf{T}$) and not against **R**. Third, in our implementation, when no destination value of the tuples in a given outer partition hashes to a particular inner partition, that inner partition is not brought into memory. As $\Delta\mathbf{T}$ decreases in size during the course of execution, the probability of such savings increases. This would not be the case if **R** or **T** were the outer relations in Seminaive and Smart, respectively, since **R** remains unchanged and **T** grows as execution progresses.

3.1.2. Timing of processing. A tuple newly inserted in $\Delta\mathbf{T}$ at some iteration can be processed at either of two times: in the next iteration, which we call *Normal* processing, or else immediately [7, 8], which we call *Immediate* processing. In Normal Seminaive, as suggested in the pseudocode in Fig. 1, the composition of $\Delta\mathbf{T}$ with **R** is completed before the new $\Delta\mathbf{T}$ is formed for the next iteration. In Immediate Seminaive, however, when a new tuple is produced, it is inserted at the end of $\Delta\mathbf{T}$ immediately so that it can be processed in the current iteration. This usually reduces the number of iterations and therefore the number of times that relations need to be accessed and read into memory from disk. In our implementation, only tuples that are generated when no page of the partition to which their destination values hash has been fully processed and flushed out to disk are immediately processed, because then immediate processing has no overhead and can only benefit performance. Also, we did not use immediate processing when duplicates are produced because they must be eliminated before they are inserted at the end of $\Delta\mathbf{T}$. With our duplicate elimination approach (at the conclusion of each iteration instead of on-the-fly) Immediate processing reduces to Normal processing. Finally, we did not consider Immediate processing for Smart because its performance is expected to be poor. The reason is that, due to the nature of Smart, a high overhead must be paid to maintain and process information about the iteration to which each tuple belongs and with what tuples it has already been joined.

3.1.3. Number of files for $\mathbf{T}$ *and* $\Delta\mathbf{T}$. In both Seminaive and Smart, the contents of $\Delta\mathbf{T}$ are always inserted into **T**. Thus, there are two options: keeping the two in separate files and always copying from one to the other, or having both reside in a single file and distinguishing the tuples of $\Delta\mathbf{T}$ by appropriately marking them in the file. When the two relations require different structures, two files are used; otherwise, only one is used. Specifically, in all nested-loops algorithms a single file is used for **T** and $\Delta\mathbf{T}$, which is hashed on the source attribute. A single file is also used in both hash-join implementations of Seminaive (Normal and Immediate), which is now hashed on the destination attribute. In the hash-join implementation of Smart, however, the file containing $\Delta\mathbf{T}$ (hashed on destination) is distinct from the file containing **T** (hashed on source). For all algorithms, **R** is an altogether separate file hashed on the source attribute.

3.1.4. Duplicate elimination. In the generic form of Seminaive and Smart (Figs 1 and 2), duplicate elimination is represented by the difference operator in the statements that update $\Delta\mathbf{T}$, where tuples already in **T** are removed from the composition results, and also when a union of the tuples in **T** with some new ones is taken (for Smart in step 5). Except for cyclic graphs, this is optional and can be ignored, possibly at the expense of efficiency. When duplicates are in fact eliminated, it is done in each iteration as part of the statement that updates $\Delta\mathbf{T}$. The following algorithm is used. The newly produced tuples from each composition (one for Seminaive, two for Smart) are written to temporary files, one for each partition used. At the end of each iteration, for each partition $\mathbf{T}_i$, a hash table on the combination of its two attributes is built in main memory. Then, the appropriate temporary files are scanned and for each new tuple in them the hash table is probed. If the tuple does not already exist, then it is inserted in $\mathbf{T}_i$, otherwise it is not. (Note that duplicates within each temporary file are also removed in this way.) If there is insufficient memory to build a hash table containing all of $\mathbf{T}_i$, the above process is done in a piece-meal fashion by moving the newly produced tuples in and out of temporary files (as in the Simple Hash-Join algorithm [18]).

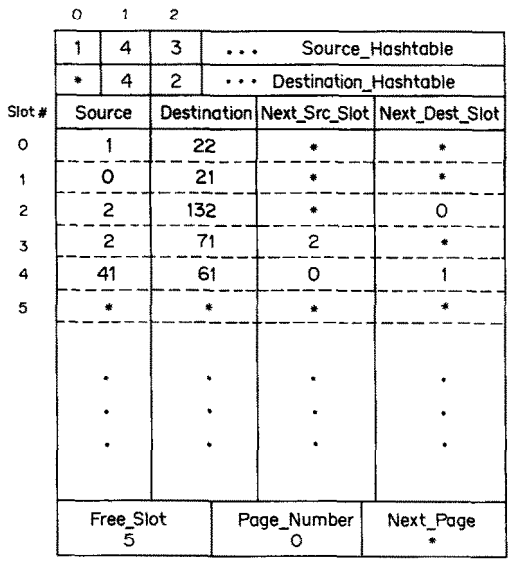

Fig. 4. Page structure for Nested-Loops Seminaive.

It is worth noting that the production of duplicates that are formed due to cycles must follow the production of a *loop arc*, i.e. an arc of the form (A, A). We take advantage of this in Seminaive, and whenever a loop arc is produced, it is inserted in **T** but not in $\Delta\mathbf{T}$. Thus, duplicates due to cycles are never produced, and the computation may terminate one iteration sooner. This also implies that, even for cyclic graphs, as long as cycles are detected in the above fashion, duplicate elimination can be ignored in Seminaive without affecting termination. Unfortunately, the same trick does not have as dramatic an effect on Smart due to the production of tuples that correspond to paths of multiple lengths from the same composition operation.

Our method for dealing with duplicates is similar to those of Valduriez and Boral [5] and Lu [8]. It differs, however, from that of Agrawal and Jagadish [6] and also Han *et al.* [7], which use a different storage paradigm for the relation and do duplicate elimination on-the-fly, i.e. as new tuples are produced. This renders the use of temporary files unnecessary, and allows the use of Immediate processing (Section 3.1.2), but it also consumes more main memory, thus leaving a smaller fraction of it available for the actual join. The trade-off between the two approaches is unclear and no definite answer exists on which one is preferable. Our decision was based on two facts. First, duplicate elimination at the end of each iteration leaves more main memory free to be used for join processing, thus resulting in join I/O savings. Second, on-the-fly duplicate elimination requires that descendents of nodes be found once for every tuple or page that is produced. This would be quite inefficient with our data structures, i.e. hash partitioning, since each search would require accessing a whole hash partition.

3.1.5. Page structure. Relations are stored in flat files. The structure of pages on disk and in the buffer pool is the same. All of our algorithm implementations deal with constant width tuples† and constant width auxiliary information per tuple. Thus, in each case, there is a specific number of slots for tuples on a page. Since there are no deletions, a page starts being filled at slot 0 and continues until it is full. The structure of an example disk page is shown in Fig. 4, where slots 0 through 4 are filled and slot 5 is the first one available. In more detail, there are 2 page structures used in the implementation of the algorithms: the *single table* structure, used by the hash-join algorithms, and the *double table* structure, used by the nested-loops algorithms. In addition to the tuples themselves, the single (respectively double) table structure contains one (respectively two) small on-page hash table for the tuples, on the source or destination attributes. These hash tables improve CPU performance by avoiding the scan of an entire page to find the tuples with a

†Independent of the width of the tuples in the original source relation, we assume that the two attributes capturing the arcs of the graph have been copied into a different relation and that their values, i.e. the node labels, have been represented with integers. This preprocessing step should almost always be taken since the subsequent space and the time savings that it provides are very significant.

particular source or destination value. Figure 4 is an example of the double table structure, with the example hashing function for both on-page hash tables being (*value mod* 10). For each hash table, tuples (slots) that hash to the same value are linked together in a list ("Next_Src_Slot" and "Next_Dest_Slot"). In Fig. 4, the first tuple with a source hash value of 1 occupies slot 4 and the next one occupies slot 0. Moreover, for the destination hash table, not all tuples are part of it; only a subset of interesting ones (in our case, only those that belong to $\Delta\mathbf{T}$) are included in this table. The reason is that, at any point, the tuples in $\mathbf{T}-\Delta\mathbf{T}$ will participate in no further joins and thus do not need an access path based on their destination values.

3.1.6. Memory management. In order to determine the effect of main memory size on the relative performance of the various algorithms and to experiment with specialized buffer replacement strategies, we decided to implement our own buffer management. This is because UNIX does not provide the capabilities of forcing page writes to disk or of limiting the amount of available main memory. If we had relied on UNIX statistics, the measurements of I/O would have been affected by hits in the UNIX buffer pool, which would have distorted the actual relative performance of the algorithms in a database context. Moreover, UNIX uses its own general purpose page replacement strategy (LRU), which is not necessarily appropriate for transitive closure computation.

We were somewhat influenced by the work of Agrawal and Jagadish [6] in dividing the buffer pool into four areas for all algorithms, as shown in Fig. 5. For every composition operation, the Delta-Load area holds pages from $\Delta\mathbf{T}$ and the Load area holds pages from the other operand of the composition (**R** or **T**). The Expansion area holds pages where newly produced tuples for **T** are placed. These pages are for the growth of **T**, when there is no duplicate elimination, and for the growth of the temporary files (Section 3.1.4) when there is. Finally, the Free area (which may be empty) holds resident pages that are available for reuse. The number of buffer pages assigned to each area and the replacement strategy are dynamic and are a function of the number of partitions, the size of each partition, and the algorithm under study. Three alternatives exist and are explained below.

Local Expansion. This scheme is used by the nested-loops algorithms. The distribution of pages in the four areas of Fig. 5 is as follows: Load contains 1 page of **R** or **T**; Delta-Load contains pages of several partitions of $\Delta\mathbf{T}$ (with one possibly being incomplete); Expansion contains 1 page for each partition of $\Delta\mathbf{T}$ that is fully or partially resident in Delta-Load; Free contains the rest. For immediate processing, Free is assigned some specific percentage of the full set of buffer pages to avoid writing new pages to disk, if at all possible, since they will be accessed in the current iteration. For future reference, this percentage is denoted by *p_Free*. For example, given a total of 10 buffer pages, the distribution in Seminaive when the first three partitions are being processed together might be as follows: 1 page in the Load area holding the current page of **R**; 6 pages in the Delta-Load area holding all of $\Delta\mathbf{T}_0$ and $\Delta\mathbf{T}_1$ and part of $\Delta\mathbf{T}_2$; 3 pages in the Expansion area—one for each partition 0, 1 and 2; and 0 pages in the Free area.

No page currently in the Delta-Load area is replaced until all of them have been joined with all of **R**. When this happens, the pages in both the Delta-Load and Expansion areas are moved to the Free area (if possible), and the next partition of $\Delta\mathbf{T}$ is then loaded in. When a page in the Expansion area becomes full in the middle of an iteration, it is exchanged with the least recently used page in the Free area, if any; otherwise, it is written out to disk and then reused.

Global Expansion. This scheme is used in the hash-join Seminaive algorithms. The distribution of pages in the four areas of Fig. 5 is as follows: Load contains all pages of one partition of **R**; Delta-Load contains 1 page of the partition of $\Delta\mathbf{T}$ that corresponds to the partition of **R** that is currently resident in the Load area; Expansion contains 1 page for each partition of the result; Free contains the rest and is associated with the *p_Free* parameter as above. Note the differences between this and the previous alternative. First, a buffer page must be reserved in the Expansion area for output to each of the partitions in the relation, since in hash-join a newly produced tuple can hash

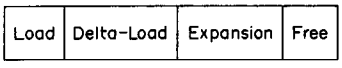

Fig. 5. The four areas of the buffer pool

Table 1. The range of algorithms studied

		Smart	Seminaive
Normal	Hash-Join	HJSm	HJSe
	Nested-Loops	NLSm	NLSe
Immediate	Hash-Join	*	HJSe-I
	Nested-Loops	*	NLSe-I

to any partition of **T**. (Recall that **T** is hashed on source, whereas the result tuples are produced in the order of a destination-based partitioning.) Second, due to hash-join, only one partition of each relation needs to be present in main memory at any one time. For example, given a total of 10 buffer pages and 3 partitions per relation, the distribution when the second partition is processed might be as follows: 4 pages in the Load area holding all of $\mathbf{R}_2$; 1 page in the Delta-Load area holding a page of $\Delta\mathbf{T}_2$; 3 pages in the Expansion area—one for each partition of **T**; and 3 pages in the Free area.

No page in the Load area is replaced until processing for that partition for the current iteration is completed. When this happens, all such pages become part of the Free area and the next partition of **R** is loaded in. The pages in the Expansion area remain allocated for the processing of the entire iteration. Full pages in the Expansion area are replaced as in the Local Expansion alternative.

Global Expansion + 1. This scheme is used by the hash-join Smart Algorithm. It is identical to Global Expansion except that, in addition to the pages for each partition of **T**, a separate page is allocated to the Expansion area for the tuples of $\Delta\mathbf{T}$. All such tuples are written to a single output buffer, regardless of the number of partitions, and are distributed to the appropriate partition of $\Delta\mathbf{T}$ at the beginning of each iteration. $\Delta\mathbf{T}$ is treated differently from **T** because there are far fewer tuples inserted into each partition of $\Delta\mathbf{T}$ than are inserted into **T**. Treating $\Delta\mathbf{T}$ similarly to **T** would result in inefficient use of main memory.

3.1.7. Space of algorithms. Our implementations of Seminaive and Smart can be divided into two classes in each of two ways. One way is according to whether the join method is hash-join or block nested-loops. The second way is according to whether Immediate or Normal processing is used. The various combinations of the above features and the names that we use to refer to each different implementation are shown in Table 1. (As mentioned in Section 3.1.2, no Immediate forms of Smart were implemented due to the expected poor performance.) The details of each algorithm's implementation are summarized in Table 2.

There are only two points that we want to clarify in Table 2. First, NLSe-I performs no duplicate elimination (Section 3.1.2), so it is essentially applicable to trees only. Second, repartitioning is a feature of the hash-join implementations that needs to be discussed. In these implementations, in order to avoid tying up too large a portion of memory for the output buffers, it is imperative to keep the number of partitions small. On the other hand, duplicate elimination is more efficient when the number of partitions is large, because then each partition is smaller and is more likely to fit in main memory. We dealt with the above conflicting constraints by making repartitioning of the initial relation an option. When no duplicate elimination is performed, there is the option of *physically* repartitioning **R** to produce the minimum number of partitions possible that still allow an entire partition of **R** to fit in main memory. The cost of this preprocessing step is expected to be far less than its benefit during the subsequent transitive closure processing. When duplicate

Table 2. Characteristics of implementations of Seminaive and Smart

		NLSe	NLSe-I	NLSm	HJSe	HJSe-I	HJSm
Number of files for T and ΔT	One	□	□	□	□	□	
	Two						□
Duplicate elimination	Yes	□		□	□		□
	No	□	□	□	□	□	□
Page structure	Single table				□	□	□
	Double table	□	□	□			
Memory management	Local expansion	□	□	□			
	Global expansion				□	□	
	Global expansion + 1						□
Repartitioning	Yes				□	□	
	No	□	□	□	□		□
Hash partition attribute (per relation)	Source	R,T	R,T	R,T	R	R	R,T
	Destination				T,ΔT	T,ΔT	ΔT

elimination is performed, there is no repartitioning because the trade-offs are unclear. Hence, repartitioning is always used in HJSe and HJSe-I for trees (these are the only graphs where HJSe-I is applicable). Because of duplicate elimination and the fact that **T** is being modified in every iteration, the optimal number of partitions is not obvious for HJSm, so no repartitioning is done in this algorithm.

3.2. Blocked Warren

This algorithm is very different from Seminaive and Smart and is thus described independently. Again, the basic issues on which decisions had to be made for the implementation of the algorithm are discussed in the subsections that follow.

3.2.1. File structure. Unlike the relations in the other algorithms, which are always stored in flat files (Section 3.1.5), **T** is stored in a B^+–tree ordered on source in Blocked Warren. We assume that the relation is originally in flat form, so it is first sorted and then the B^+–tree is built in a bottom up fashion. Our B^+–tree implementation is a simple one. In the leaf pages of the tree, there is no secondary ordering on destination—ordering is determined solely by the source value of the tuple. Also, when there are more tuples with the same source value than can fit in a page, an overflow page is allocated. Each such page holds destination values for a single source value, thus making extremely efficient use of space within the page since the source value does not have to be repeated with each destination value.

3.2.2. Main memory data structures. Unlike Seminaive and Smart, there are several different data structures occupying main memory that are significant enough in size to be counted as memory requirements of the algorithm. First, there are the buffered pages of the B^+–tree, whose structure remains identical to its form on disk. Second, when brought into memory from disk, the information in the B^+–tree leaf pages is copied into *descendent lists*, one for each source value. These descendent lists are structured as blocks of destination values linked together in a doubly-linked list. These lists are needed when a source node is part of the diagonal or an off-diagonal block that is being processed. Third, descendent lists of nodes in the diagonal block are indexed with *descendent hash tables*. These are used to avoid adding duplicate descendents and to efficiently search for descendents of a given node in the diagonal block. A separate hash table is kept for each node in the diagonal block. These are transitory structures: when a node becomes part of the diagonal block, a hash table is constructed; when the contents of the diagonal block change, the hash table is destroyed. Each hash table is not a single large structure, but is instead composed of a main table and several overflow blocks; therefore, its size depends upon the number of descendents of the corresponding source node.

3.2.3. Memory management. For Blocked Warren, main memory is divided into three areas. First, a specific number of buffers are reserved solely for B^+–tree pages, including both internal pages and leaves. For future reference, this number is denoted by *p_Btree*. Second, at any time, the descendent list of a single off-diagonal element must be kept in main memory, so we reserve enough space for the largest such list. Specifically, we reserve enough space to store up to the total number of nodes that appear as destinations but not as sources of tuples in **R**. This is obviously an upper bound on the actual size of the largest descendent list and is determined at the time when **R** is being loaded in order for the B^+–tree to be built. Third, the remaining buffer pages are used by the descendent lists of the diagonal block elements and their hash tables.

One question regarding the use of the third area is the fraction of it that is occupied by the descendent lists (and hash tables) at the beginning of the first or second pass of the algorithm, since some of it must be left free for growth. Confirming the results of Agrawal and Jagadish [6], we have observed that a large part of the growth of the (diagonal element) descendent lists usually occurs during the first pass of the algorithm. As a consequence, we loaded only a small percentage *p_first* of the third area at the beginning of the first pass, whereas we increased that limit to a larger percentage *p_second* at the beginning of the second pass.

Except for the root of the tree, which is pinned in memory, the rest of the B^+–tree buffer pages are managed by an LRU strategy. Descendent lists are placed on a free list when not being directly used, so they can be replaced by other descendent lists using an LRU strategy as well. However, if they are reaccessed while they are still in memory, they are reclaimed from the free list instead of being rebuilt from B^+–tree pages. In the event that the third area becomes full and the

descendent lists in it need to grow more, the last node of the diagonal block is removed, thus freeing up the space occupied by its descendent list and hash table. In this sense, dynamic partitioning is realized (Section 2.2.3).

3.2.4. Size of diagonal blocks. As mentioned above, following Agrawal and Jagadish [6], we allowed for dynamic changes in the size of the diagonal block. When there is insufficient memory to accommodate the reading of a descendent list for an off-diagonal element or to accommodate the addition of elements to the descendent list of a diagonal element, the size of the diagonal block is reduced. This is done by removing the last node from the diagonal block and freeing the space occupied by the corresponding descendent lists. The first node of the next diagonal block is always the node following the last node of the previous block that has been fully processed.

4. TESTBED

We experimented with all seven versions of the algorithms described in the previous section, i.e. the 6 versions of Seminaive and Smart shown in Table 1 and Blocked Warren. They were implemented in C on a VAXstation 3200 running UNIX. The VAXstation had 8 Megabytes of main memory, of which 0.5 Megabyte was used by user programs and data. Its file system block size was 2 kbytes, so the file page size and the buffer page size in our implementation were also chosen to be 2 kbytes. All reported experiments were run while no other users were on the machine. However, because we did our own buffer management, with UNIX doing its own file system buffering underneath, the UNIX-provided elapsed times are not directly meaningful. In our experiments, we thus relied on UNIX-provided CPU times and our own counting of disk reads and writes based on the implemented page replacement strategy and the amount of available memory in each case. We combined the two to produce a performance metric that we refer to as the *aggregate time*, which is the sum of the measured CPU time plus 40 msec for each disk read or write performed. We arrived at the 40 msec value after experimentation with the disk of the machine that we used for our experiments.

There are several interesting parameters that affect the performance of the algorithms. They can be divided into parameters of the algorithm implementations and parameters of the data. These two parameter classes are discussed in the following subsections. The third subsection discusses the process of graph generation that was used to produce the test data for the experiments.

4.1. Parameters of the algorithm implementations

There are two interesting parameters of the algorithm implementations that are meaningful in most cases: the *number of partitions*, i.e. the number of hash buckets in the data file, and the *number of buffer pages*. We varied the number of buckets from 1 up to a point beyond which performance was seen to be deteriorating. We also varied the number of buffer pages from 20 to 250. In addition to the above, we varied two more parameters that are of interest in specialized cases to investigate their effect on performance: for data forming acyclic graphs, there is the option to eliminate duplicates or not, and for HJSe, there is the option to repartition the initial relation or not. Finally, there are four additional parameters, mentioned in Section 3, that were assigned specific values based on a series of initial experimental results. Specifically, we set $p\_Free = 10\%$, $p\_Btree = 5$, $p\_first = 15\%$ and $p\_second = 80\%$. The last two numbers are the same ones used by Agrawal and Jagadish [6], as our preliminary experiments with these parameters essentially confirmed their results regarding good settings for them.

4.2. Parameters of the data

Without loss of generality, all relations used in our experiments contained integer node identifiers randomly generated in a specific range. (Even if a given relation is not in this form, it can be transformed accordingly in a single pass [5].) For any specific setting of the values of the parameters described below, all algorithms under comparison were run on the same input graph.

Using graph terminology, we experimented with all three interesting types of data: trees, acyclic graphs and general graphs. Graphs are usually characterized by *generating* parameters, i.e. parameters used to generate them, such as the *number of nodes* and *outdegree* (or *branching factor*). Unfortunately, depending on the specifics, graphs with similar values for these parameters can have

dramatically different transitive closures, and therefore the various algorithms can exhibit very different behavior on them. Several such examples have been given by Lipton and Naughton [19]. In this study, we used the following parameters: the size |**R**| of **R** in tuples, the size |**T**| of **T** in tuples, the number P of tuples produced by Seminaive when duplicate elimination is performed (not including the original tuples of **R**), and the depth d of the graph. (Similar parameters are used by Valduriez and Boral [5].) As the forthcoming results show (Sections 5.3 and 6.3), using these parameters yields a clearer understanding of the trade-offs because of their direct effect on performance. For convenience, we occasionally describe graphs using their generating parameters.

4.3. Graph generation process

For most experiments, graphs were generated randomly based on desirable values of their generating parameters. As mentioned above, the specific data values representing the graph nodes were chosen randomly as well. All trees with which we experimented were regular, so their generation was straightforward. Acyclic graphs were generated by randomly chosing the successors of each node among all other nodes that were greater than the said node in some arbitrary order (essentially the order of node creation). Cyclic graphs were generated by randomly chosing the successors of each node among all other nodes.

For the experiments that study the effect of graph characteristics on the performance of the algorithms, we used different techniques for generating graphs. This was done because we were interested in controlling the values of parameters that could not be influenced directly. Depending on the specific experiment, one of two approaches was taken. In the first approach, several random graphs were generated; out of these a carefully selected subset was chosen so that all graphs had approximately equal values (less than a 5% difference) for all but one parameter and were significantly different in their values of the remaining parameter. For example, to study the effect of |**R**| on performance when |**T**| and d are kept constant, many graphs were generated; a subset with similar values for |**T**| and d and significantly different values for |**R**| was chosen for the experiments.

In the second approach, a family of graphs was generated where each graph consisted of multiple components. The largest component was generated randomly, whereas the rest were regular trees used to control the values of the desired parameters. The first graph of the family was generated randomly based on the desired properties in the experiment. The remaining graphs were then generated in two steps: (a) random modifications were made to the original graph based on the desired characteristics of the generated graph; (b) because this modification usually altered more than one of the graph's characteristics, an appropriate number of disconnected tree components were then added to compensate for the alterations of all but one of the affected parameters. Determining the appropriate number and type of trees was straightforward, as we used regular trees whose characteristics are easily computable. For example, to study the effect of |**T**| on performance when P and d are kept constant, a single graph **R** was generated originally and its transitive closure **T** was computed. Several graphs were then generated by removing different percentages of the arcs in **T**–**R** from **T**. To keep P constant, the resulting graphs were each complemented with trees whose transitive closure had the same number of arcs as the ones deleted from **T**. Thus, P remained constant while |**T**| increased. Also, the depth d remained constant due to another tree that was part of all graphs which was deeper than all other components. Using similar techniques, we were able to generate graphs that were different only in one of the directly relevant parameters and similar in all others.

5. COMPUTING THE FULL TRANSITIVE CLOSURE OF TREES

When processing trees, no algorithm produces duplicates. Hence, except where otherwise noted, we have run all experiments reported in this section without performing any duplicate elimination. In addition, HJSe was run without any repartitioning. In all results presented in this and the following two sections, the aggregate time is measured in seconds.

5.1. General trends

5.1.1. Performance of Seminaive and Smart. Table 3 shows a typical example of the number of I/O operations and aggregate time (in seconds) of the various flavors of Seminaive and Smart when

Table 3. Performance results for all algorithms on a regular binary tree ($b = 2$) of depth $d = 11$

Algorithm	I/O requests	Aggregate time (sec)
NLSe	879	63.7
NLSe-I	633	57.8
NLSm	1068	79.9
HJSe	689	47.2
HJSe-I	583	42.4
HJSm	434	27.3
Blocked Warren (topological order node numbering)	799	297.0
Blocked Warren (random node numbering)	5857	704.0

50 buffers are used. For each algorithm, these results were obtained by dividing relations into the optimum number of partitions†. The specific example is for a regular binary tree ($b = 2$) of depth $d = 11$. The initial relation for this tree contains $|\mathbf{R}| = 4094$ tuples, and the transitive closure contains $|\mathbf{T}| = 40{,}962$ tuples. The exact sizes of **R** and **T** in pages vary with the algorithm, since disk pages in the different algorithms have slightly different structures and therefore hold slightly different numbers of tuples. With a single partition, the size of **R** is between 23 and 25 pages and the size of **T** is between 220 and 250 pages.

This example is typical of the relative performance of the algorithms on trees and indicates the following trends. Regarding the join method, nested-loops is generally the poorer performer. There are three basic reasons for this. First, in the nested-loops implementations, each $\Delta\mathbf{T}_i$ must be joined with *every* $\mathbf{R}_i$ or $\mathbf{T}_i$, whereas in the hash-join implementations, each $\Delta\mathbf{T}_i$ must only be joined with its corresponding $\mathbf{R}_i$ or $\mathbf{T}_i$. When memory is in short supply, the above implies that nested-loops requires **R** or **T** to be read from disk many times per outer iteration. On the other hand, as long as there is enough memory to hold one partition of $\mathbf{R}_i$ or $\mathbf{T}_i$ plus one page from $\Delta\mathbf{T}_i$, hash-join is very efficient since **R** or **T** are read from disk only once per iteration. Second, because of the different CPU requirements of the algorithms, different page structures have been adopted for them. Specifically, the single table structure is used for file pages in the hash-join algorithms, whereas the double table structure is used for file pages in the nested-loops algorithms. Hence, more tuples can be stored in each page for the hash-join algorithms than for the nested-loops algorithms. This implies that more pages are produced in nested-loops which, in turn, implies fewer hits in the buffer pool. Note, however, that the single table page structure would negatively affect the performance of nested-loops (because there would be no fast way to access tuples based on their destination value), increasing its total aggregate time even more. Third, specifically for Smart, a single file is used in NLSm to hold **T** and $\Delta\mathbf{T}$, whereas different files are used in HJSm. In NLSm, this has the effect that the tuples of $\Delta\mathbf{T}$ are not necessarily stored in adjacent slots within **T**, thus requiring more processing time and/or consuming more buffer pages in each iteration than in HJSm.

Regarding the timing of processing, the Immediate paradigm shows advantages for both the nested-loops and hash-join algorithms, which confirms the results of Lu [8] and Han *et al.* [7]. That is, as expected, a savings in I/O results when newly produced tuples are immediately reused.

Regarding the basic transitive closure algorithm, Smart and Seminaive rank differently depending on the employed join method. For nested-loops, Seminaive outperforms Smart. This is because Smart deals with much larger relations and therefore traverses longer hash chains than Seminaive. Since chain traversal is a very common operation in nested-loops (quadratic in the number of partitions), it dominates performance with respect to both CPU time and I/O and Smart is penalized. For hash-join, Smart outperforms Seminaive. This is because, although Smart still deals with larger relations, chain traversal is less common than in nested-loops. Thus, the advantage of Seminaive in searching shorter hash chains becomes less marked and the fewer iterations of Smart becomes the dominant feature. In this example, the overall best performance is exhibited by HJSm.

†These numbers do not include the cost of partitioning the initial relations if they are not already partitioned. An initial set of experiments showed that this is negligible compared to the overall cost, so we ignored it by assuming that relations are always appropriately partitioned.

5.1.2. Performance of Blocked Warren. The performance results for Blocked Warren on a regular tree with $b = 2$ and $d = 11$ are given in Table 3. In this case, Blocked Warren is by far the poorest performer. The primary reason for this is that, in trees, the root and other nodes close to it tend to have a large number of descendents. The size of these descendent lists significantly limits the number of elements that can be part of the diagonal block. This forces the algorithm to iterate over the for each loops of Fig. 3 many times, thus exhibiting poor performance especially with respect to the aggregate time.

An interesting point to note is that the performance of Blocked Warren is very sensitive to the numbering of the nodes in the graph. In Table 3, two results are shown for the same tree, one when the nodes of the tree are numbered in topologically sorted order, and another when they are randomly numbered. The performance of Blocked Warren on the latter tree is more than a factor of 2 worse than on the former in aggregate time. Even at its best, Blocked Warren is still worse than any of the implementations of the other algorithms, and the others are insensitive to the numbering of the nodes.

5.2. Effect of parameters of algorithm implementations

In the subsections below, we discuss how the performance of the various algorithms is affected by the number of partitions, duplicate elimination, and repartitioning. The effect of the number of buffers is similar for both trees and non-tree graphs and is discussed in Section 6.2.2.

5.2.1. Number of partitions. Figures 6 and 7 show a typical example of how the number of I/O operations and aggregate time (in seconds) for the various flavors of Seminaive and Smart vary as a function of the number of partitions into which the relations are divided. These figures are for a regular binary tree ($b = 2$) of depth $d = 11$, i.e. for the same data discussed in Section 5.1. Thus, the minima exhibited by each algorithm in Fig. 7 correspond to the results presented in Table 3.

In general, for all algorithms, the I/O portion of the transitive closure cost increases with the number of partitions. This is because one partition corresponds to the lowest fragmentation possible, i.e. to the smallest number of pages. In addition, ΔT is often small enough so that it fits in main memory, and so does **R**. Thus, I/O performance reaches its optimum at a single partition. This general pattern is broken by a few irregularities, which are due to statistical peculiarities of the hash functions used that occasionally result in excessive fragmentation.

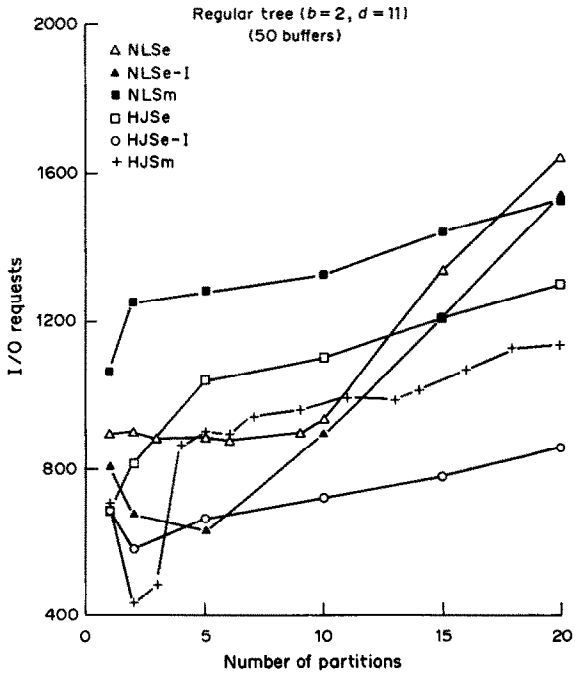

Fig. 6. I/O operations vs number of partitions for Seminaive and Smart applied on trees.

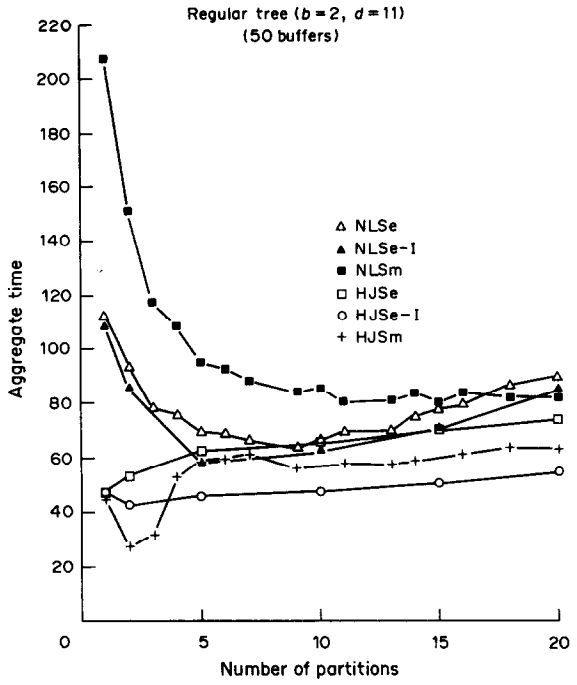

Fig. 7. Aggregate time vs number of partitions for Seminaive and Smart applied on trees.

On the other hand, for almost all algorithms, the best aggregate time is obtained with a small number of partitions, usually between 2 and 10, in Fig. 7. This behavior is easily explained by the I/O behavior in the area to the right of the optimum (as all algorithms are I/O bound in that area). This is not the case, however, in the area to the left of the optimum, where the nested-loops algorithms are heavily CPU bound. The smaller the number of partitions in a relation, the larger the size of each partition, i.e. the longer the chains that need to be searched are when probing the relation, and therefore, the larger the amount of CPU time spent. This determines performance up to a certain point, beyond which fragmentation becomes a problem and I/O cost becomes dominant. As verified by Fig. 7, this trend is more pronounced for Smart than for Seminaive. This is because Smart probes **T** whereas Seminaive probes **R**, and **R** is a much smaller relation.

5.2.2. Duplicate elimination. There are situations where no or very few duplicates exist (as is the case with trees), but that fact is unknown when the transitive closure is computed, so duplicate elimination must be performed albeit unnecessarily. Table 4 shows the aggregate time of HJSe and HJSm with and without duplicate elimination, for a regular binary tree ($b = 2$) of depth $d = 11$. Again, the results are obtained by dividing relations into the optimum number of partitions. (Nested-loops implementations are now shown because they are very slow with duplicate elimination.) Observe the significant increase in cost for Seminaive and Smart with duplicate elimination over the case where no duplicate elimination is attempted. While both algorithms still perform better than Blocked Warren, the difference is considerably smaller. Comparing the two, Smart is again faster than Seminaive for the same reason (the difference in the number of iterations required for termination). Also, note that Blocked Warren remains almost unaffected by incorporating duplicate elimination. This is because the cost of the algorithm is dominated by the significant amount of processing that is necessary due to the large size of the descendent lists; the extra overhead of checking for nonexistent duplicates is relatively small in comparison.

Table 4. Optimal aggregate time (sec) for Seminaive and Smart with and without duplicate elimination applied on trees

Algorithm	With duplicate elimination	Without duplicate elimination
HJSe	159.0	47.2
HJSm	130.0	27.3
Blocked Warren	312.0	297.0

5.2.3. Repartitioning in hash-join Seminaive. Figure 8 shows a typical example of the effect of repartitioning on the performance of HJSe as a function of the original number of partitions. The specific curves (with and without repartitioning) are again for a regular binary tree ($b = 2$) of depth 11 with 50 buffers. It is clear that repartitioning has a significant pay-off for HJSe, essentially allowing it to always achieve its optimal performance independent of the original number of partitions. Thus, when no duplicate elimination is required, repartitioning should always be part of HJSe. (Recall that in all other implementations of our algorithms, no repartitioning is performed because the optimal number of partitions is unclear.)

5.3. Effect of parameters of data

Recall that the important graph parameters for comparing the various transitive closure algorithms are $|\mathbf{R}|$, $|\mathbf{T}|$, P and d. For trees, no duplicates are produced, i.e. $|\mathbf{T}| = P + |\mathbf{R}|$, so the interesting parameters are reduced to a set of three (ignoring P). In each of the following subsections, we keep two of those parameters relatively constant and vary the third one so that its effect on performance when all other things are equal becomes clear.

We should note that randomly generating different graphs with exactly equal $|\mathbf{R}|$ or $|\mathbf{T}|$ values is difficult. Thus, as mentioned in Section 4.3, we experimented with graphs whose values for these parameters were less than 5% different. This difference between the actual values was small enough so that it did not affect the quality of the results. We should also mention that it is extremely hard to generate trees with equal values of $|\mathbf{R}|$ and $|\mathbf{T}|$ that differ significantly in d. Moreover, d affects the performance of algorithms similarly for both trees and non-trees. Thus, we do not present any experiments that vary d while holding $|\mathbf{R}|$ and $|\mathbf{T}|$ constant; the conclusions of Section 6.3.3 dealing with non-trees are applicable in this case as well.

In Section 5.1.1, we concluded that hash-join is in general superior to nested-loops as a join method. In addition, we demonstrated that Blocked Warren is not competitive with the other algorithms on trees where duplicate production and elimination is not an issue. Thus, in this section, we only present results for HJSe and HJSm.

We should also point out that, for these experiments, the values for the parameters that remain constant impose limits on the values that the varied parameter can take on. Thus, most figures in this section (and Section 6.3) do not present data for very small values of the varied parameter because there are no graphs with such values.

5.3.1. Size of R. The effect of $|\mathbf{R}|$ on the behavior of the algorithms when $|\mathbf{T}|$ and d remain constant is shown in Fig. 9. The specific example shown is for trees with $d = 10$ and $|\mathbf{T}| \approx 18{,}300$. As expected, the cost of HJSe almost monotonically increases with $|\mathbf{R}|$. **R** is one of the two operands

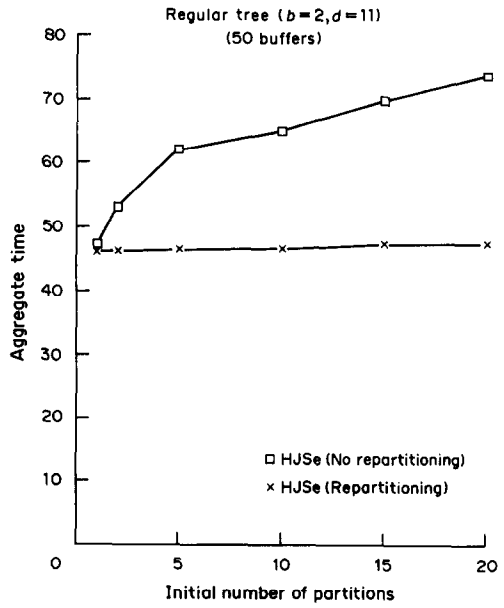

Fig. 8. Effect of repartitioning on the performance of HJSe.

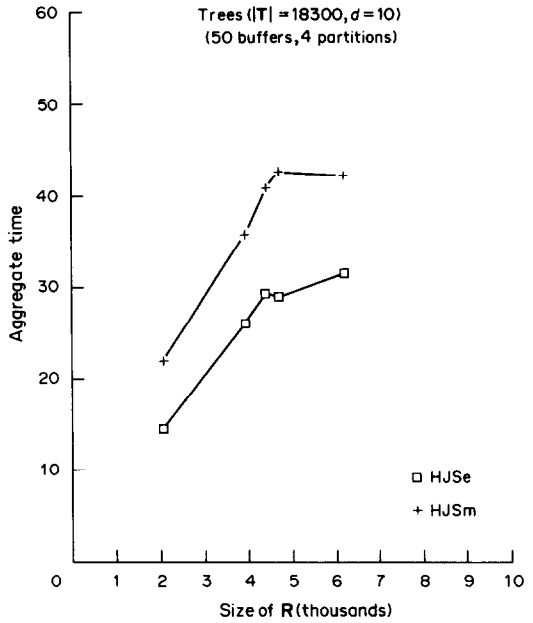

Fig. 9. Effect of |**R**| when |**T**| and d are constant.

for all joins, so its increase in size significantly affects performance. The cost of HJSm, however, displays a somewhat different behavior. It increases faster than HJSe in the beginning, until it reaches a maximum, and then it remains relatively flat. This is explained as follows. First, increasing |**R**| introduces more tuples in T from the beginning as well, which makes the algorithm more costly. In addition, since |**T**| remains constant, parts of the tree tend to become shallower as |**R**| increases. That is, although the depth of the tree remains the same, the average distance of a node from the root decreases. This implies that tuples tend to be produced for the first time in earlier iterations, making early iterations more expensive and late ones cheaper. To the left of the maximum, the cost increase in the early iterations dominates the cost decrease in the later ones. Beyond a certain point, pretty much all of the tuples are generated in the first iterations, so the cost flattens out.

5.3.2. Size of T. The effect of |**T**| on the behavior of the algorithms when |**R**| and d remain constant is shown in Fig. 10. The specific example is for trees with $d = 10$ and approximately $|\mathbf{R}| = 4400$. As expected, the cost of both algorithms is monotonically increasing with |**T**|, with HJSm being affected more severely. This is due to the fact that both operands of the joins of HJSm increase in size, whereas only one of them does for HJSe. In addition, this difference is enhanced by the fact that HJSm performs fewer iterations than HJSe. That is, the impact of the increase in the number of tuples that need to be processed on each iteration of HJSm is more dramatic than on those of HJSe.

6. COMPUTING THE FULL TRANSITIVE CLOSURE OF NON-TREE GRAPHS

We treat both acyclic and cyclic graphs together because, for the most part, no major difference was observed between the two in any aspect of our experiments. Duplicate elimination is the key to the performance of all algorithms in non-tree graphs. As we have already mentioned, although termination of both Seminaive and Smart is guaranteed for acyclic graphs even without duplicate elimination, the performance implications of not eliminating duplicates can be devastating because of the enormous number of tuples generated and the extra work that needs to be done to process them.

Table 5 contains data detailing the importance of duplicate elimination. Specifically, for Seminaive and Smart, we show the number of tuples that are inserted in **T**, the total number of tuples produced (excluding those initially in **R**), and the necessary number of iterations, both with and without duplicate elimination, for several acyclic graphs. We use P' to denote the total number of tuples produced to distinguish it from P, which was explicitly defined to be the number of tuples

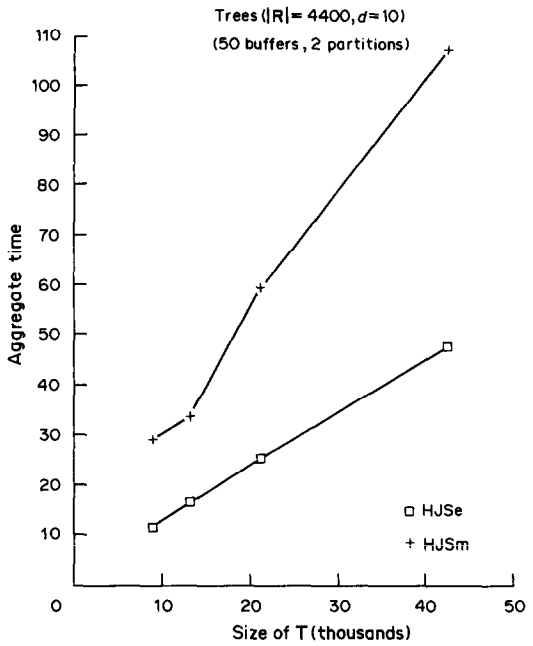

Fig. 10. Effect of |T| when |R| and d are constant.

produced by Seminaive when duplicate elimination is performed (Section 4.2). We observe that a very large number of duplicates can be produced even from a graph that has a very small transitive closure. The graph with 20 nodes and a branching factor of 10 is an excellent example of this; the transitive closure has only 184 tuples, but when duplicates are not eliminated, about 200,000 tuples are produced. (As a result, the execution time for HJSe increases by more than a factor of 200—from 1 sec to almost 4 min—when 150 buffers are used!) Also, note that Smart and Seminaive produce the same number of tuples when no duplicate elimination is done. This is because, in that case, both algorithms produce one tuple for every path in the graph, so their differences in execution have no effect on tuple production. When duplicate elimination is done, however, Seminaive produces fewer tuples than Smart. This is because one iteration of Smart is equivalent to several iterations of Seminaive, which implies that Seminaive eliminates duplicates more often and avoids possible multiplicative effects [20].

Because of these observations, duplicates were always eliminated in our experiments with non-tree graphs. This implies that no immediate algorithms are applied on such graphs (Section 3.1.2). The results of our non-tree experiments are described in the following subsections.

6.1. General trends

6.1.1. Performance of Seminaive and Smart. Table 6 shows an example of the cost of the various algorithms applied on non-tree graphs. The particular results are for all algorithms operating on a cyclic graph with 100 nodes and a branching factor of $b = 10$ whose depth is $d = 4$. Fifty buffers are used and relations are divided into the optimum number of partitions for each algorithm. **R** contains 1000 tuples, while **T** contains 10,000 tuples and occupies approximately 55–60 pages. We

Table 5. Effect of duplicate elimination

Graph description		Algorithm	No duplicate elimination			Duplicate elimination		
N	b	class	|T|	P'	Iterations	|T|	P'	Iterations
20	3	Seminaive	1227	1173	9	130	280	4
20	10		191,324	191,179	16	184	960	3
100	2		5693	5496	11	1263	2230	7
100	3		156,710	156,416	20	2283	6259	8
20	3	Smart	1227	1173	4	130	280	3
20	10		191,324	191,179	5	184	965	2
100	2		5693	5496	4	1263	2482	3
100	3		156,710	156,416	5	2283	8599	4

Table 6. Performance results for all algorithms on a cyclic graph of $N = 100$ nodes and degree $b = 10$

Algorithm	Aggregate time (sec)
NLSe	132.0
NLSm	478.6
HJSe	130.2
HJSm	438.8
Blocked Warren (topological order node numbering)	53.3

present an extreme case as an example to clearly distinguish these results from those on trees. The example is extreme in that **T** is a complete graph. Qualitatively, for a large range of graphs with which we experimented, the relative performance of the algorithms was similar to that in Table 6. Only on graphs that were very tree-like did the results become closer to those of Table 4. In Table 6, it is clear that there is a definite superiority of Seminaive over Smart in this case. In addition, hash-join is slightly better than nested-loops for Smart, whereas the two join methods perform very similarly for Seminaive.

The reason for the very strong similarity of NLSe and HJSe in performance in this case is duplicate detection and elimination. This consumes a large percentage of their execution time. Since the join method does not affect the number of duplicates produced or the cost per tuple in duplicate elimination, their similarity in performance is natural. Moreover, duplicate elimination requires the loading of **T** into main memory. Except for very small graphs, this causes most other pages to be flushed from memory. For both HJSe and NLSe, it is thus likely that much or all of **R** has to be read from disk at each iteration, making the cost of the remaining part of their operation very similar as well.

The reasoning in the previous paragraph explains the similarity of NLSm and HJSm in performance as well. These two algorithms, however, exhibit a larger difference between their Seminaive counterparts. This is because, for NLSm, the $\Delta\mathbf{T}_i$ tuples are not adjacent within **T**, whereas they are placed in a small separate file for HJSm. Thus, probing in **T** to look for matches is less efficient in NLSm than in HJSm.

Duplicate elimination is also the reason for the difference in performance between Seminaive and Smart. Table 7 shows the number of duplicate tuples produced by the two algorithms when duplicate elimination is done for several different graphs. (We use P and P_{sm} to denote the number of tuples produced by Seminaive and Smart respectively. Since this number does not include the tuples initially in **R**, the number of duplicates produced by Seminaive is equal to $P - (|\mathbf{T}| - |\mathbf{R}|)$ and similarly for Smart.) Because of the condensation of several iterations into one and the fact that duplicate elimination happens just once per iteration, Smart generates many more tuples than Seminaive. In addition, the duplicate elimination cost per tuple is the same for both algorithms. Hence, since the overall cost is dominated by that of duplicate elimination, Table 7 makes it no surprise that Seminaive outperforms Smart in this case.

6.1.2. Performance of Blocked Warren. Table 6 shows the performance of Blocked Warren as well. As indicated in the table, Blocked Warren is by far the most efficient algorithm in this case. Its cost is also dominated by duplicate detection and elimination, but it produces fewer duplicates in general, thus excelling in performance. We should mention, however, that there do exist non-tree graphs on which Blocked Warren is inferior to the other algorithms. In particular, these are graphs that have a very large transitive closure that is computed almost in its entirety in the first pass of the algorithm (Fig. 3). Such an example is a cyclic graph with 400 nodes and branching factor of $b = 10$ whose transitive closure is complete. Aggregate time, I/O operations, and the number of

Table 7. Effect of duplicate production on Smart and Seminaive

Graph description			Seminaive		Smart	
N	b	$\lvert\mathbf{T}\rvert$	$P - \lvert\mathbf{T}\rvert + \lvert R\rvert$	Iterations	$P_{sm} - \lvert\mathbf{T}\rvert + \lvert\mathbf{R}\rvert$	Iterations
100	3	9500	18,715	9	204,649	4
100	10	4235	33,780	6	57,894	3
200	5	10,574	39,331	8	84,247	4
200	15	17,305	218,950	6	563,344	3

Table 8. Performance characteristics for graph with complete transitive closure

Algorithm	Aggregate time (sec)	I/O requests	Number of produced tuples (P)
HJSe	2154.2	25,002	1,596,000
Blocked Warren	3031.9	12,225	51,453,000

tuples produced for this particular graph using 150 buffers are given in Table 8 for Blocked Warren and HJSe. Blocked Warren has inferior performance on such graphs primarily because of the CPU time spent on duplicate elimination. After the first pass of Blocked Warren, the majority of tuples are already in **T**. Thus, the number of duplicates produced in the second pass is large, incurring a significant CPU cost (I/O cost is kept relatively low since duplicate elimination is performed on-the-fly). Seminaive produces many duplicates also, but this number is relatively limited. In particular, as shown in Table 8, Seminaive generates approximately 1.5 million tuples, whereas Blocked Warren generates approximately 50 million. This difference explains the poor performance of Blocked Warren.

In general, as will be discussed further in Section 6.3.3, the performance of Blocked Warren is affected by the depth of the graph. In particular, when all other things are equal, Blocked Warren is relatively slow on shallow graphs. This is roughly explained as follows. For a given tuple of **T**, the sequence of tuples of **R** that need to be composed to generate it is shorter in shallow graphs than in deep ones. Hence, the probability that the node numbering will be in the appropriate order for the tuple to be produced in the first pass of Blocked Warren is higher in shallow graphs than in deep ones, as fewer nodes in shallow graphs need to have numbers in the correct order. Thus, shallow graphs tend to produce more tuples in the first pass, which has the compound effect of producing even more tuples in the second pass and requiring them to be compared against larger descendent lists.

As a complement to the above experimental evidence, we present another example below and show analytically the number of tuples produced by each algorithm. Consider the graph shown in Fig. 11(a), whose adjacency matrix is shown in Fig. 11(b). For this graph, $|\mathbf{T}| = n^2$ and $d = 2$, i.e. it is a shallow graph with a large, actually a complete, transitive closure. To emphasize the point, we have numbered the graph nodes in the worst possible way for Blocked Warren. Following the algorithm step by step yields that the number of tuples produced in the first pass of Blocked Warren is equal to $(n^2 - 2)(n - 1)/2$, and the corresponding number for the second pass is equal to $n^2(n - 1)/2$, for a total of $(n^2 - 1)(n - 1)$ tuples produced. On the other hand, Seminaive generates a total of $2(n - 1)^2$ tuples, which is $O(n)$ fewer than Blocked Warren.

We should also emphasize the sensitivity of Blocked Warren to the numbering of the nodes. If in the above graph we exchange the numbers for nodes 1 and n, we obtain the best case for blocked Warren. The number of tuples produced in the first pass then drops to $n - 1$, with the number produced in the second pass dropping to $n(n - 1)$, for a total of $(n^2 - 1)$. This is a dramatic drop in the number of tuples produced by Blocked Warren, to the point where asymptotically it is half the corresponding number for Seminaive.

In conclusion, although in the majority of cases Blocked Warren is the algorithm of choice, the point needs to be made that there are shallow graphs with large transitive closures where Blocked Warren generates many more duplicates than Seminaive, thus leading to inferior performance for

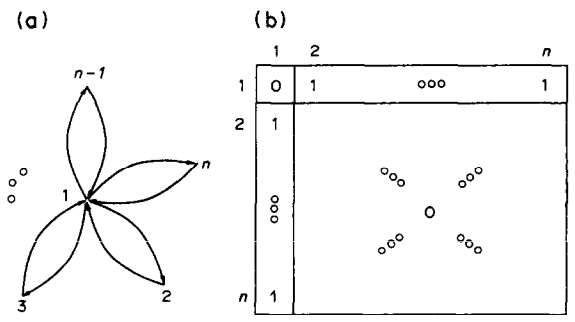

Fig. 11. Graph where Blocked Warren performs poorly.

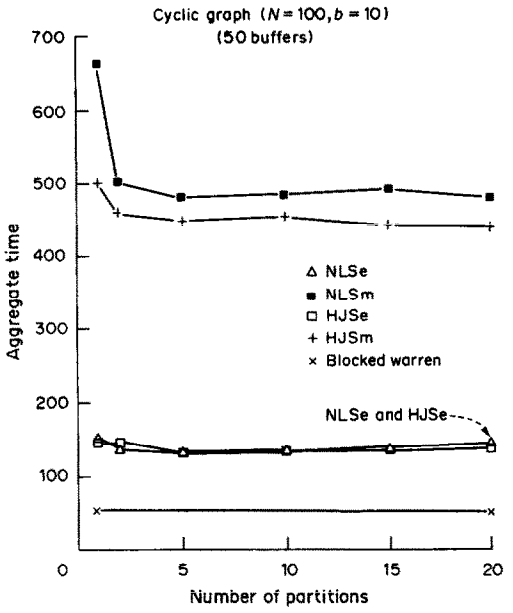

Fig. 12. Aggregate time vs number of partitions for Seminaive, Smart and Blocked Warren applied on non-tree graphs.

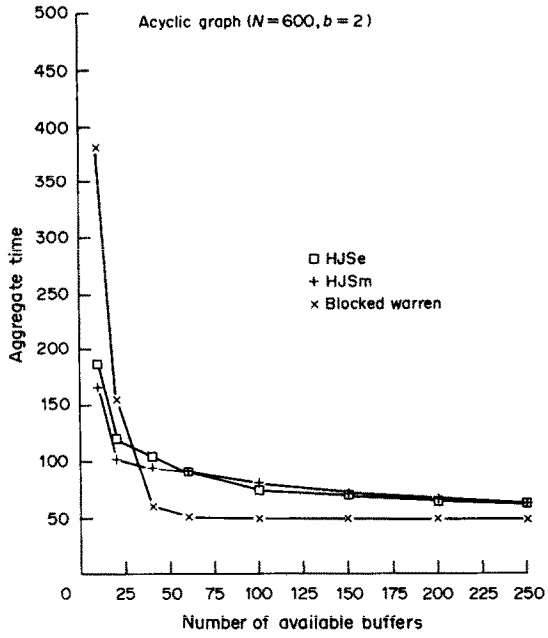

Fig. 13. Aggregate time vs available memory for Seminaive, Smart and Blocked Warren.

Blocked Warren in such cases. Moreover, whether or not Blocked Warren indeed behaves poorly is a question whose answer is node numbering dependent.

6.2. *Effect of parameters of algorithm implementations*

6.2.1. Number of partitions. Figure 12 displays data for the typical cost of the various algorithms applied on non-tree graphs as a function of the number of partitions. The specific example is for the same graph in Section 6.1, i.e. a cyclic graph with 100 nodes and a branching factor of $b = 10$ (with depth $d = 4$), and for 50 buffers. Thus, the minimum point of each curve in Fig. 12 corresponds to the associated entry in Table 6. Moreover, since the performance of Blocked Warren is not affected by the number of partitions, it is shown as a straight line for the sake of comparison. As we can see in Fig. 12, the number of partitions does not affect the performance of any algorithm on non-tree graphs as significantly as it did for trees. This is due to duplicate elimination, which dominates the cost and flattens out the curves over a very wide range of the number of partitions.

6.2.2. Number of available buffers. Figure 13 is a typical figure of the aggregate time for HJSe and Blocked Warren for non-tree graphs as a function of the number of available buffers. (The results for nested-loops implementations are not shown because they are similar to those of the hash-join implementations.) The specific example is for an acyclic graph with 600 nodes and a branching factor of $b = 2$ whose transitive closure contains $|\mathbf{T}| = 17{,}457$ tuples. **T** occupies 100 pages for HJSe and 117 pages for Blocked Warren. The difference is primarily due to B^+–tree page splits throughout the course of the algorithm, which leave many pages of the result just slightly over 50% utilized.

The performance of HJSe and HJSm degrades as memory is reduced, but the degradation becomes significant only at quite small buffer sizes. On the other hand, Blocked Warren is much more sensitive to the amount of main memory available. With large memory, Blocked Warren is clearly superior to Seminaive and Smart. As memory becomes smaller, however, performance degrades more severely for Blocked Warren; it becomes worse than the other algorithms when memory size drops to about 30% of the size of **T**. The memory size sensitivity of Blocked Warren is primarily due to duplicate elimination, as checking for duplicates requires that descendent lists be in main memory. A small number of buffers in combination with this requirement forces the size of the diagonal blocks to decrease, which has negative effects on I/O and CPU time that have already been discussed (Section 5.1.2).

6.3. Effect of parameters of data

As in Section 5.2, the important graph parameters are |**R**|, |**T**|, P and d. For non-tree graphs, all four of these parameters are in principle independent and affect the performance of the algorithms differently. Except for extreme cases of very small or very large graphs, however, the role of |**R**| is minor. Moreover, it is almost impossible to construct multiple graphs that have equal values for the remaining three parameters and differ significantly in the value of |**R**|. Thus, we have concentrated on |**T**|, P and d, examining the effect of each one while keeping the other two constant. For the reasons explained above, the value of |**R**| was ignored in these experiments. The results are organized as in Section 5.2. Again, we only discuss the hash-join algorithms, since it is clear from Fig. 12 that they are either superior or equivalent to their nested-loops counterparts. Finally, as in Section 5.3, most figures in this section do not present data for very small values of the varied parameters because there are no graphs with such values.

6.3.1. Size of T. The effect of |**T**| on the behavior of the algorithms when P and d remain constant is shown in Fig. 14. The specific example is for acyclic graphs with $d = 40$ and $P \approx 77{,}000$. We should first note that HJSm is superior to HJSe due to the large value of d. On shallower graphs, although the following discussion on the effect of |**T**| remains valid, HJSe is superior to HJSm as indicated in Table 6. The cost of HJSe increases with |**T**| in an approximately linear fashion. This is because, for all joins, the size of the one operand, Δ**T**, increases due to the increase in |**T**|. Duplicate elimination also becomes more costly since the same number of produced tuples must be checked against a larger **T**. HJSm displays the same behavior up to a certain point, beyond which its cost becomes stable. The reason for the difference is the following. The fact that P remains constant while |**T**| increases implies that the number of duplicates, i.e. useless tuples, produced by HJSe decreases. This is true for HJSm as well, but in a more dramatic way, since HJSm generates more duplicates than HJSe. Hence, as |**T**| increases, HJSm becomes more competitive. This also explains the relative flatness of the curve for HJSm on the high end of |**T**| values. Recall that, for the graphs used in this experiment, HJSe produces the same number of tuples (P). This is not the case for HJSm, however; in fact, as |**T**| increases, HJSm produces fewer tuples overall (i.e. a higher number of duplicates disappear). This results in the relative flatness of the curve. Finally, the cost of Blocked Warren is also a monotonically increasing function of |**T**|. In this case, the number of tuples produced by Blocked Warren is not controlled by keeping P constant, and it in fact increases with the increase in |**T**|, thus significantly affecting the total cost. In addition, main memory becomes inadequate to hold the needed descendent lists and hash tables beyond a certain point,

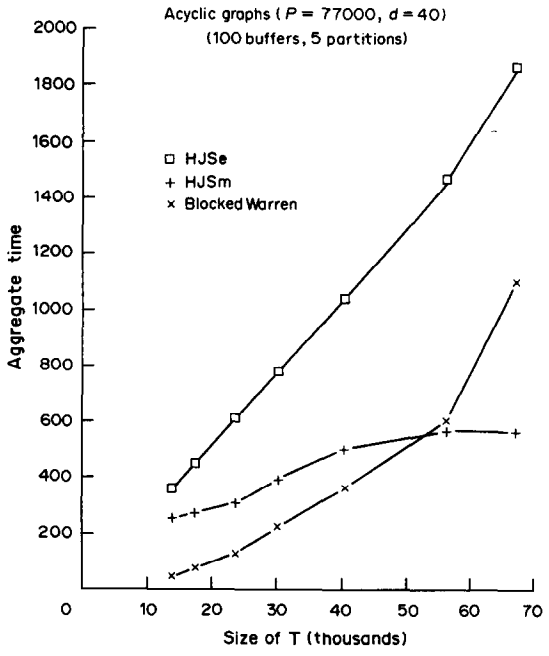

Fig. 14. Effect of |**T**| when P and d are constant.

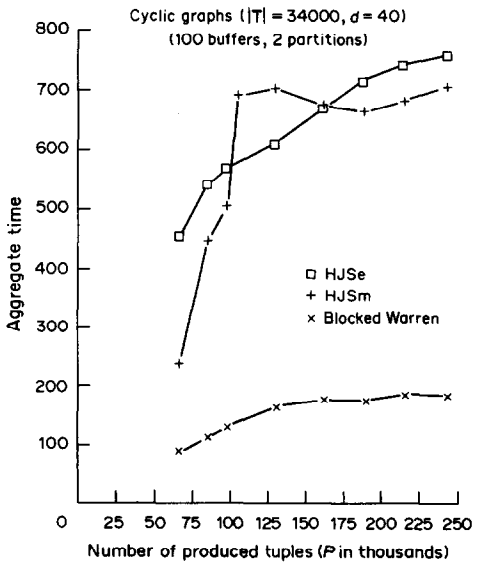

Fig. 15. Effect of P when |**T**| and d are constant.

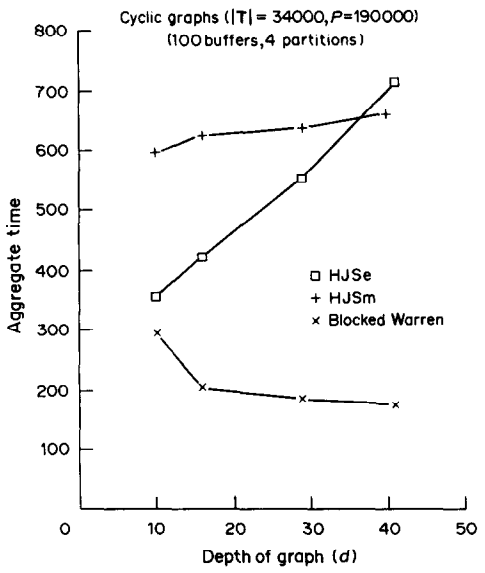

Fig. 16. Effect of d when |T| and P are constant.

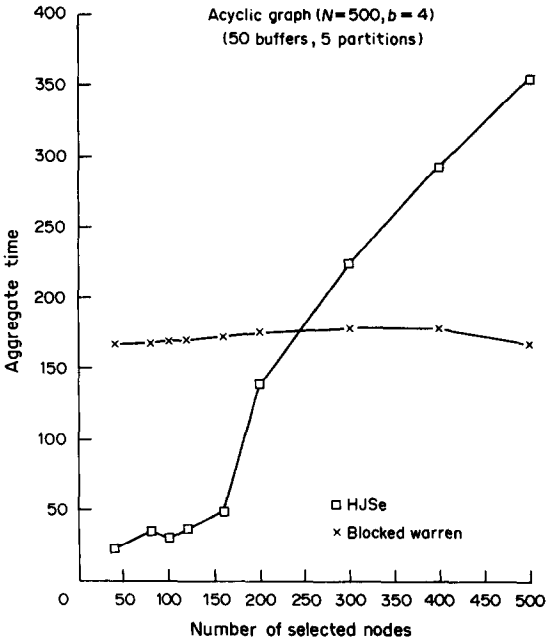

Fig. 17. Aggregate time vs number of selected nodes for Seminaive and Blocked Warren.

and this has two effects. First, I/O cost increases. Second, the size of the diagonal blocks decreases in order for the descendent lists of their nodes to fit in main memory. This implies an increase in the number of internal iterations for Blocked Warren, causing an associated increase in the overall cost of the algorithm, as discussed in Section 5.1.2. The compound effect of all of the above factors can be observed at the far right end of the Blocked Warren curve in Fig. 14.

6.3.2. Number of produced tuples. The effect of P on the behavior of the algorithms when |T| and d remain constant is shown in Fig. 15. The specific example shown is for acyclic graphs with $d = 40$ and $|\mathbf{T}| \approx 34{,}000$. The behavior of HJSe and HJSm is similar to the case for trees when |T| and d were kept constant while |R| was varied. The cost of HJSe monotonically increases with P. The more duplicates that are produced, the more expensive the duplicate elimination process becomes. The cost of HJSm increases much faster than HJSe in the beginning, until it reaches a maximum, beyond which it remains relatively flat. The explanation for this behavior is similar to that of the corresponding case for trees. HJSm inherently produces more duplicates than HJSe and is therefore affected more severely by the increase in P. In addition, this increase needs to be dealt with in many fewer iterations in HJSm. Thus, the increased cost of duplicate elimination is the dominant factor in the initially steep increase in the cost of HJSm. On the other hand, since |T| remains constant, tuples tend to be produced for the first time in earlier iterations of HJSm (or HJSe) as P increases, making the earlier iterations even more expensive while the later ones become cheaper. To the left of the maximum, the cost increase in the early iterations dominates the cost decrease in the later ones. Beyond a certain point, the vast majority of the tuples are generated in the first few iterations, so the cost flattens out. The small variations in the flat area are affected by the specific way that the cost of each iteration changes based on the specific graph and other parameter choices. Finally, as expected, the cost of Blocked Warren is affected little by the increase in P. This is because of the completely different nature of the algorithm, which generally produces fewer duplicates than HJSe and does so in a completely different fashion. Initial increases in P have some impact on the behavior of the algorithm, since there are more tuples that need to be processed, but beyond a certain point increasing P does not imply an increase in the number of tuples produced by Blocked Warren and its cost is therefore stabilized.

6.3.3. Depth of graph. The effect of d on the behavior of the algorithms when |T| and P remain constant is shown in Fig. 16. The specific example is for cyclic graphs with $P \approx 190{,}000$ and $|\mathbf{T}| \approx 34{,}000$. HJSe has a steep increase in its cost as d increases due to the extra overhead of the additional joins and duplicate elimination steps that must be performed. The fact that the total

number of tuples that need to be looked at remains the same implies that most individual iterations become cheaper, but this gain is dominated by the increase in the number of iterations. Specifically, the last few iterations end up joining **R** with relations that have approximately the same number of tuples as **T** and are therefore quite expensive. HJSm is also increasing in cost for the same reasons as above, but the increase is much slower due to its logarithmic number of iterations. Unlike these two algorithms, the cost of Blocked Warren is monotonically decreasing in d. This is related to the comments in Section 6.1.2, where graphs on which Blocked Warren performed poorly were presented. Given that $|\mathbf{T}|$ and P are constant, as the graph becomes deeper, i.e. as d increases, the probability that a tuple will be generated during the first phase of Blocked Warren decreases. This has the double effect of leading Blocked Warren to process fewer tuples in the second pass and needing to compare them against fewer tuples to check for duplicates. Hence, although the number of tuples produced by HJSe (P) remains constant, the number of tuples produced by Blocked Warren decreases, which results in its overall cost decrease.

7. COMPUTING A SELECTED PART OF THE TRANSITIVE CLOSURE OF GRAPHS

On transitive closure queries that involve a selection, Smart is not applicable because it cannot make full use of the selection (Section 2.2.2), and NLSe is always inferior to HJSe due to the overall superiority of hash-join over nested-loops. Thus, for such queries, we experimented only with HJSe and Blocked Warren. HJSe can take advantage of the selection by accessing only the part of the graph that is relevant to the query. In contrast, Blocked Warren must first compute the complete transitive closure and then apply the selection.

Figure 17 shows typical aggregate times for Blocked Warren and HJSe as a function of the number of nodes selected by the query. The specific examples are for an acyclic graph with 500 nodes, 1990 arcs, and a branching factor of $b = 4$. The full transitive closure of this graph contains 43,733 arcs. Both algorithms were run with 50 buffers, and HJSe was run with five partitions for the relations.

As expected, when few nodes are selected, HJSe is superior to Blocked Warren. The surprising observation, however, is that the percentage of nodes that need to be selected for Blocked Warren to be superior is very high. In the example of Fig. 17 this threshold is nearly 50% of the nodes in the graph. In general, the thresholds that we observed over a wide range of experiments were always greater than 35%. Some might argue that most queries will be more selective than that, which implies that Seminaive may almost always be the algorithm of choice in practice.

It is worth noting that Seminaive is relatively flat up to a certain number of selected nodes, i.e. 30% of all nodes in the graph associated with Fig. 17. Beyond that point its cost increases rapidly and becomes worse than that of Blocked Warren. The reason for the initial flatness is that the relevant relations are small and all operations can be performed in main memory. The knee in the curve occurs at the point where enough nodes are selected so that I/O becomes a necessity.

8. COMPARISON TO RELATED WORK

A significant body of work exists on computing the transitive closure of a binary relation in a disk-based environment. In the previous sections, we have referred to several articles that deal with transitive closure algorithms that are relevant to our study. In addition to those, work has been done on using graph-braversal for transitive closure computation [4, 21], combining Blocked Warren with depth-first search [21, 22], maintaining the transitive closure in compressed form without significant losses in performance [23, 24], and computing the transitive closure in parallel [10, 11, 25]. In the following subsections, we compare the results of our study with those of related previous studies and discuss the reasons for cases where our results differ from those of related studies.

8.1. Effect of duplicate elimination and immediate processing

The positive effect of immediate processing was noted first by Lu [8], who incorporated it into an implementation of Seminaive. Han *et al.* [7] analyzed the expected performance of several

different algorithms for recursive query evaluation, especially algorithms that are very similar to Seminaive for transitive closure computations. The Han *et al.* study concluded that duplicate elimination and immediate processing both improve performance and that hash-bash implementations outperform B^+–tree based ones. The significant role of duplicates in the performance of all algorithms was also noted by Agrawal and Jagadish [6].

The results of our study confirm those of previous studies with respect to duplicate elimination and immediate processing. Given the different approaches that underly these distinct studies, we can infer that these conclusions are rather universal.

8.2. Seminaive vs Smart

In contrast to the findings of this study, previous studies have shown that, in most "interesting cases", Smart outperforms Seminaive. This was the conclusion of Ioannidis [3], Valduriez and Boral [5] and Lu [6]. Ioannidis studied the performance of the two algorithms on trees (where no duplicates are produced), and Smart was observed to be superior to Seminaive on all types of trees of all sizes except very large ones. Valduriez and Boral found that Smart was superior to Seminaive when the depth of the graph was above a certain small threhold. Below that threshold they found Seminaive to be superior, but the difference was small. Finally, Lu essentially confirmed the results of Ioannidis.

In Section 6.1.1, we have shown that Seminaive generally dominates Smart for non-tree graphs, usually by a large margin, except when the depth of the graph is large. The difference in conclusions between this study and its predecessors can primarily be attributed to the different implementations of the algorithms that were used or assumed in each study. Immediate processing, specialized buffer replacement strategies, duplicate elimination, and hash-based join methods are all present in our study, whereas most of them were missing from previous studies. Hashing was the basic data structure used in our implementations, whereas different structures were used in other studies, e.g. join indices [5]. Moreover, our study involved a real implementation of the algorithms, whereas the earlier studies have relied on simplified simulations [3, 8] or theoretical analysis [5]. Some of these studies used a model for the data that is not realistic for many graphs, e.g. a model where the selectivity factors of the joins performed in any two consecutive iterations of Seminaive have a constant ratio [5], while others showed results on trees alone without performing any duplicate elimination [3]. The much richer set of techniques used in our implementations and the higher degree of completeness of our experiments make us confident that the conclusions of this study should override those of the previous ones with respect to the relative performance of Seminaive and Smart.

8.3. Seminaive and Smart vs Warren

We have already mentioned that Blocked Warren was first introduced by Agrawal and Jagadish [6]. The results of that study showed Blocked Warren to be superior to Smart in all cases. Despite improved implementations of Smart (and Seminaive), our study essentially confirmed that result in general. Nevertheless, we have also identified classes of graphs where the opposite holds. These are shallow graphs with large transitive closures. Although such graphs are not expected to be common, recognizing them is useful in understanding the relative merits of the algorithms.

Lu *et al.* [9] compared the performance of Blocked Warren and two versions of Smart. Their study was different from ours and from that of Agrawal and Jagadish in two respects. First, a different adaptation of Warren to a database environment was assumed. The approach taken there involved partitioning the rows of the adjacency matrix based on hash values without any provision for dynamic repartitioning. Second, the algorithms were not actually implemented; Lu *et al.* proposed an implementation and then analyzed it to estimate performance. Their results indicated that Smart is affected by small memory less than Blocked Warren, and that Smart is actually superior to Blocked Warren when the size of the transitive closure is much greater than the size of memory. This has been confirmed by our experiments (Section 6.2.2). They also examined the effect of graph characteristics on the performance of the two algorithms. Most of their results vary more than one parameter at a time, so it is hard to compare them directly with ours. Nevertheless, to the extent possible, the conclusions of the two studies seem to be mutually agreeable.

A unique contribution of our study is the examination of the relative performance of the algorithms in the presence of various degrees of selection. Previous studies have experimented only with one end of the spectrum (queries with no selection), although some of them have also informally discussed the other end of the spectrum (very selective queries). No previous study, however, has attempted to quantify the effect of query selectivity over a wide range of values on the relative performance of these algorithms. We have investigated this issue and have demonstrated that Seminaive is superior to Blocked Warren for selections of surprisingly large subsets of the nodes in the graph. The specific value of the fraction of nodes that are selected beyond which Blocked Warren becomes superior to Seminaive depends on the graph, but it was always observed to be above 35% in our experiments.

9. SUMMARY

In this paper, we have investigated alternative implementations of known algorithms for transitive closure and we have evaluated their relative performance. The results obtained complement those of previous studies and offer new insights on how these algorithms should be implemented for improved performance. In particular, the primary contributions of this paper are the following. First, several alternatives for the implementation of Seminaive and Smart have been investigated. Specialized data structures and buffer management strategies have been proposed to improve the performance of each different algorithm. Also, our experiments have demonstrated that (i) hash-join is the preferred join algorithm for transitive closure for both Seminaive and Smart, (ii) processing tuples as soon as they are generated and are still in memory is beneficial whenever it is applicable, (iii) repartitioning is always beneficial for hash-join Seminaive and (iv) Seminaive and Smart become more competitive compared to Blocked Warren as the buffer pool size decreases. Second, the effects of several data characteristics on the behavior of these transitive closure algorithms has been analyzed. Graph parameters that directly affect algorithm performance have been identified, and a relatively novel approach has been employed to independently control each such parameter in order to separately study its impact on algorithm performance. Several experiments have shown that no algorithm is universally superior, i.e. for each algorithm examined there are graphs for which the algorithm is the most efficient. In particular the major conclusions can be qualitatively summarized as follows:

(i) As expected, increasing the size of **R**, the size of **T**, or P makes all algorithms perform worse. Interestingly, when increasing the values of P or $|\mathbf{R}|$ while keeping the other parameters constant, the graphs tend to be shallower, so the costs of Seminaive and Smart remain stable beyond a certain value of the varied parameter. This is because most of the transitive closure is computed within a few iterations.

(ii) The more tree-like the graph is (i.e. the smaller the difference between P and $|\mathbf{T}|$ is), the more competitive Seminaive and Smart are compared to Blocked Warren, especially when its descendent lists are large (since they limit the size of the diagonal blocks of Blocked Warren). As more and more duplicates are produced, however, Blocked Warren becomes the algorithm of choice. Also, Smart becomes more competitive compared to Seminaive when the graph is tree-like, as the duplicate reduction is more significant for Smart than for Seminaive.

(iii) The deeper the graph is, the more competitive Smart is compared to Seminaive. Also, Blocked Warren becomes even more competitive on deep graphs for the reasons explained in Section 6.1.2. On the other hand, for shallow graphs, when $|\mathbf{T}|$ is large, Seminaive appears to be the algorithm of choice.

Finally, the performance of Seminaive and Blocked Warren in the presence of selections has been studied. As expected, since Blocked Warren cannot take advantage of a selection, Seminaive is superior for very selective queries. Surprisingly, our experiments have shown that the percentage of the graph nodes that need to be selected for Blocked Warren to be superior to Seminaive is rather large (for all graphs tested, it was greater than 1/3). This implies that in the majority of realistic transitive closure queries with selection, Seminaive is likely to be the preferred strategy.

Acknowledgements—Y.E.I. was partially supported by a grant from IBM and by the National Science Foundation under Grant IRI-8703592. M.J.C. was partially supported by a grant from IBM and by the National Science Foundation under Grant IRI-8657323.

REFERENCES

[1] A. Aho and J. Ullman. Universality of data retrieval languages. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, San Antonio, TX, pp. 110–117 (1979).
[2] M. M. Zloof. Query-by-Example: a data base language. *IBM Syst. J.* **16**(4), pp. 324–343 (1977).
[3] Y. E. Ioannidis. On the computation of the transitive closure of relational operators. In *Proc. 12th Int. VLDB Conf.*, Kyoto, Japan, pp. 403–411 (1986).
[4] Y. E. Ioannidis, R. Ramakrishnan and L. Winger. Transitive closure algorithms. In *Proc. 14th Int. VLDB Conf.*, Long Beach, CA, pp. 382–394 (1988).
[5] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Proc. 1st Int. Conf. on Expert Database Systems*, Charleston, SC, pp. 197–208 (1986).
[6] R. Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *Proc. 13th Int. VLDB Conf.*, Brighton, England, pp. 255–266 (1987).
[7] J. Han, G. Qadah and C. Chaou. The processing and evaluation of transitive closure queries. In *Proc. 1988 Int. Conf. on Extending Database Technology*, Venice, Italy, pp. 49–75 (1988).
[8] H. Lu. New strategies for computing the transitive closure of a database relation. In *Proc. 13th Int. VLDB Conf.*, Brighton, England, pp. 267–274 (1987).
[9] H. Lu, K. Mikkilineni and J. P. Richardson. Design and evaluation of algorithms to compute the transitive closure of a database relation. In *Proc. 3rd Int. Conf. on Data Engineering*, Los Angeles, CA, pp. 112–119 (1987).
[10] R. Agrawal and H. V. Jagadish. Multiprocessor transitive closure algorithms. In *Proc. Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, pp. 56–66 (1988).
[11] P. Valduriez and S. Khoshafian. Transitive closure of transitively closed relations. In *Proc. 2nd Int. Conf. on Expert Database Systems*, Tysons Corner, VA, pp. 177–185 (1988).
[12] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. In *Proc. 1990 ACM-SIGMOD Conf. on the Management of Data*, Atlantic City, NJ, pp. 44–53 (1990).
[13] F. Bancilhon. Naive evaluation of recursively defined relations. *On Knowledge Base Management Systems—Integrating Artificial Intelligence and Database Technologies* (Edited by M. Brodie and J. Mylopoulos), pp. 165–178. Springer, NY (1986).
[14] A. Guttman. New features for relational database systems to support CAD applications, PhD Thesis, University of California, Berkeley, CA (1984).
[15] R. Kung, E. Hanson, Y. E. Ioannidis, T. K. Sellis, L. Shapiro and M. Stonebraker. Heuristic search in data base systems. In *Expert Database Systems, Proc. 1st Int. Workshop* (Edited by L. Kerschberg), pp. 537–548. Benjamin/Cummings, Menlo Park, CA (1986).
[16] H. S. Warren. A modification of Warshall's algorithm for the transitive closure of binary relations. *CACM* **18**(4), pp. 218–220 (1975).
[17] S. Warshall. A theorem on boolean matrices. *JACM* **9**(1), pp. 11–12 (1962).
[18] L. D. Shapiro. Join processing in database systems with large main memories. *ACM TODS* **11**(3), pp. 239–264 (1986).
[19] R. J. Lipton and J. F. Naughton. Estimating the size of generalized transitive closures. In *Proc. 15th Int. VLDB Conf.*, Amsterdam, The Netherlands, pp. 165–171 (1989).
[20] S. Ganguly, R. Krishnamurthy and A. Silberschatz. An analysis technique for transitive closure algorithms: A statistical approach. In *Proc. 7th Int. Conf. on Data Engineering*, Kobe, Japan, pp. 728–735 (1991).
[21] B. Jiang. A suitable algorithm for computing partial transitive closures in databases. In *Proc. 6th Int. Conf. on Data Engineering*, Los Angeles, CA, pp. 264–271 (1990).
[22] R. Agrawal and H. V. Jagadish. Hybrid transitive closure algorithms. In *Proc. 16th Int. VLDB Conf.*, Brisbane, Australia, pp. 326–334 (1990).
[23] R. Agrawal, A. Borgida and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. 1989 ACM-SIGMOD Conf. on the Management of Data*, Portland, OR, pp. 253–262 (1989).
[24] H. V. Jagadish. A compressed transitive closure technique for efficient fixed-point query processing. In *Proc. 2nd Int. Conf. on Expert Database Systems*, Tysons Corner, VA, pp. 209–223 (1988).
[25] J. Cheiney and C. de Maindreville. A parallel strategy for transitive closure using double hash-based clustering. In *Proc. 16th Int. VLDB Conf.*, Brisbane, Australia, pp. 347–358 (1990).