in: Proc. 2nd Symposium on Mathematical Fundamentals of Database Theory, Visegrad, Hungary, June 1989, pp. 415-428

Transitive Closure and the LOGA+-Strategy for its Efficient Evaluation

W. Yan

Central South University of Technology, Changsha, China
current address: Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 6750 Kaiserslautern, West Germany

N. Mattos

Department of Computer Science, University of Kaiserslautern

Abstract

One of the key problems when extending relational database query languages to include deductive capabilities, is to provide them with efficient methods for answering recursive queries. During the last few years many algorithms have been proposed to deal with transitive closure computation of a relation. In this paper, we discuss some important criteria for developing transitive closure algorithms. After presenting these issues, we describe an algorithm for transitive closure computation and show some results of performance measurements comparing several algorithms.

1. Introduction

In the last few years the support of deductive capability in database systems (DBS) has been extensively required by the increasing number of so-called "non-standard" applications. One of the most promising approaches to support this requirement is to extend DBS to include the whole functionality underlying the theoretical foundation of first order logic. In so doing, a user of such a Deductive DBS (DDBS) [GMN84] can descriptively express his queries as well as his integrity constraints in first order logic. A traditional DBS user, on the other hand, has to implement such deductions by means of application programs embedding DB-calls.

In general, DDBS consist of a set of base relations (Extensional DB: EDB) with explicitly stored tuples and a set of rules defining virtual relations (Intensional DB: IDB). The tuples of a virtual relation are derived from the clauses of the IDB and from the tuples contained in the EDB. Such a derivation can be achieved by the dynamic view mechanism of relational DBS, however, only to a certain extent: this mechanism cannot handle recursively defined virtual relations.

In the last few years this problem has received growing interest in the DB research area, leading to several proposals of strategies to efficiently evaluate recursion in logical queries. An overview and analysis of some of the strategies can be found in [BR86, HL86, Ya87]. Most of the existing strategies have been developed to deal with all types of recursion. However, due to the existence of different types of recursion, it is almost impossible to develop a universal strategy which handles all types of recursions and at same time works efficiently. For this reason, researchers have concentrated on a special type of recursion - transitive closure problem (TC-problem), since most of the recursion problems occurring in real world applications are of this type [HQC88, LMR87]. Furthermore, some large classes of recursive queries, e.g. linear and nonlinear recursive queries, can be translated into a TC-problem proceeded and followed by relational algebra operations [JAN87, ZY87, IW88].

In this paper, we give the reader an introduction to the TC-problem. Section 2 discusses some issues that are important criteria for developing algorithms to evaluate the TC of a relation. Section 3 presents an algorithm according to these criteria. Section 4 provides the results of some performance measurements comparing different recursion strategies.

2. The TC-Problem and its Evaluation

2.1 Definition

Using notation of Horn clauses the TC-problem can be defined as follows:

CLOSURE(X,Y) ← BASIS(X,Y)
CLOSURE(X,Y) ← CLOSURE(X,Z), BASIS(Z,Y)

Where BASIS(X,Y) is the base relation, whose transitive closure should be computed and held in the virtual relation CLOSURE(X,Y) after evaluation. The arguments X, Y and Z have the same value domain.

In addition to this Horn clauses notation, the transitive closure can be represented by means of a directed graph. In this representation, we can view a binary relation as a directed graph, where the nodes express attribute values of the relation and edges express the tuples. If we consider BASIS as a base graph, then its transitive closure is also a directed graph with the same set of nodes as BASIS. An edge in form <a, b> means that the node b is able to be reached from the node a in the base graph.

The recursion depth of a relation is defined as the length of the longest path in the base graph .

2.2 Primitive Solution for TC-Problem

The simplest and most direct algorithm for evaluating a transitive closure of a relation is the Naive Evaluation [Ba86, BMSU86]. In Naive Evaluation, the rules are compiled into an iterative program in which a composition (that is, a join with a subsequent projection) is carried out to generate new tuples. They are then added to the result relation CLOSURE, which has previously been initiated with BASIS. The process is repeated until no new tuple can be generated. The following relational algebra program (RA-program) describes the Naive algorithm [BR86]. For simplicity, attributes of the relations have been omitted.

CLOSURE := BASIS
While "generating new tuples "
begin
    AUX := CLOSURE o BASIS
    CLOSURE := CLOSURE + AUX
end

Here o stands for composition and + for the union operation. AUX is an auxiliary relation which contains the newly generated tuples that are appended stepwise to CLOSURE in each iteration.

Although Naive Evaluation may provide the correct transitive closure of a relation, it gives a very bad performance. In the following, we discuss some problems that arise when processing the TC-problem by means of Naive Evaluation and analyze them, in order to point out some improvements that have influenced the development of our efficient strategy.

- Avoidance of redundant workRedundant work occurs when rules are evaluated more than once based on the same data, i.e., operations are repeated senselessly [Ba86, BR86]. Primarily, there are two possible causes for it. One is the existence of cyclic data that directs the evaluation back to its start point (this will be discussed in some detail later). The other lies in the awkwardness of an algorithm. This is just the case of Naive Evaluation, since the computation of an i-th iteration repeats entirely what was done by the last iteration. Following the lines of the RA-program given above we illustrate this problem as follows. "i" is used for expressing a loop index.

  In the i-th iteration we have:

1) $AUX_i = \mathbf{CLOSURE_{i-1} \circ BASIS}$

2) $CLOSURE_i = CLOSURE_{i-1} + AUX_i$

and in the i+1-th iteration

3) $AUX_{i+1} = CLOSURE_i \circ BASIS$

4) $CLOSURE_{i+1} = CLOSURE_i + AUX_{i+1}$

from 1) and 2):

5) $CLOSURE_i = CLOSURE_{i-1} + \mathbf{CLOSURE_{i-1} \circ BASIS}$

from 2) and 3):

6) $AUX_{i+1} = (CLOSURE_{i-1} + AUX_i) \circ BASIS$

$= \mathbf{CLOSURE_{i-1} \circ BASIS} + AUX_i \circ BASIS$

from 4) and 6):

7) $CLOSURE_{i+1} = CLOSURE_i + \mathbf{CLOSURE_{i-1} \circ BASIS} + AUX_i \circ BASIS$

We see clearly that the part $CLOSURE_{i+1}$ o BASIS, which was generated when computing AUX_i and was added to CLOSURE in the i-th iteration, is again computed when evaluating AUX_{i+1}. This part is then unnecessarily appended to $CLOSURE_{i+1}$ causing an expensive elimination of duplicate tuples.

It is obvious that such redundant work directly affects the efficiency of a strategy. For this reason, it should be avoided as far as possible.

- Efficient evaluation of queries with constants

  Up till now we have viewed the TC-problem as a whole TC-evaluation, e.g., "ancestors of all persons contained in a base relation". However, most queries arising in practice refer only to parts of the transitive closure, e.g. "ancestors of Melina". Such queries are denoted queries with constants or partial transitive closure which involve selections on the underlying relations. The set of data which is necessary for evaluating each one of them is called relevant data of the query.

  There are basically two different methods for answering such queries. The first one, which is also the method that is used by Naive Evaluation, computes the whole transitive closure and adds an extra selection at the end in order to pick out the relevant part. The other method confines the computation to the relevant data from the very beginning by using the query constants to generate a kind of "query-related transitive closure" [HQC88], thereby avoiding superfluous computations. This method corresponds to a very important heuristic used in query optimization, "perform selections as early as possible ", which creates better results in most cases.

  In fact, constants represent a powerful filter that qualifies only those data relevant to the query. There are two possible types of such filters for a query with constants: the start filter which lays down a start point for the evaluation of the query (e.g. "ancestors of John"), and the stop filter, which specifies a condition for a break criterion, i.e., the evaluation should be stopped as soon as the condition is reached. A combination of both kinds of filter is also possible by determining whether there exists a path between two specified nodes (e.g. "is Mary an ancestor of John?").

  When evaluating queries containing constants, the second method taking both filters into account is certainly the most appropriate due to its better performance.

- Handling set oriented queries

  Set oriented queries are those involving more than one constant, e.g. " ancestors of John and Melina", so that they can be expressed as combinations (and, or, etc.) of isolated queries containing constants, this corresponds to finding nodes that are reachable from a set of nodes rather than from a single specified node.

  There are three ways to evaluate them. The first is to compute the whole TC and then to perform a selection dependent on the combination of constants. This method which is used in Naive Evaluation is obviously inefficient, because the constants are not taken into consideration from the very beginning. In the second method, a query for each constant is separately processed and then the results combined in an appropriate manner. It shows better performance than the first method, but is, however, still inefficient, since the combination of constants is considered in a later stage. The third method which is in all cases more efficient pays attention to all constants and their combination, so that the evaluation is performed only once right from the beginning.

- Processing cyclic data

  Cycles are paths which originate and end at the same node of a graph. They occur very often in data of practical applications as for example in traffic connections. Here, one can generally leave a city and return to it by following different paths.

  The processing of cyclic data creates an additional demand on a strategy. Since the evaluation returns to the starting point, a strategy may repeat the evaluation, thereby doing a considerable amount of redundant work, and even getting into a endless loop. To prevent this, one can either identify cycles in advance and handle them appropriately or develop a strategy, that terminates evaluation of cycles once a node is reached twice.

  It is quite difficult and expensive to identify cyclic data in advance. Furthermore, new checks are required every time changes on the data are performed. For this reason, the second solution seems to be the only practical one. A strategy should not be restricted to specific classes of data, but should work correctly on all types of data structures. Actually, this problem is not presented by Naive Evaluation, since the "newly generated tuple" are determined during the evaluation. By doing this, cyclic data leads the evaluation to the starting point, but generates no new tuples.

- Fast convergence

  The convergence factor of a strategy can be expressed by the number of iterations necessary to compute a transitive closure.

  Observe the RA-program of Naive Evaluation, the search for reachability in a graph is performed by relational DB-operations in a breadth first fashion. Therefore, when computing the transitive closure, the attained path length normally grows monotonically with the number of iterations. In Naive Evaluation, for example, the computation progresses by one step along the recursion depth in each iteration. In this sense, it presents a linear convergence factor. In other words, for computing a transitive closure, Naive Evaluation needs as many iterations as the recursion depth of the base relation.

  However, the convergence factor of a strategy does not always have to be linear. A possibility for improving the efficiency of a strategy is to increase its convergence factor thereby speeding up its evaluation process.

3. The LOGA$^+$ Strategy

3.1 The Algorithm

Based on the discussion of section 2, we now present a new strategy called LOGA$^+$. In this strategy, high efficiency is achieved mainly by speeding up the evaluation process, as well as by dealing with queries containing constants appropriately. In our algorithm, a temporary relation, DELTA, which is joined with itself during each iteration, has been introduced. This join doubles the recursion depth of DELTA (1, 2, 4, 8, 16, . . .) in each iteration, so that the evaluation process reaches the recursion depth very quickly. Additionally, before entering the iteration process, a selection on the basis of the query constants is performed so that only the relevant data is qualified and further propagated (use of a start filter). Evaluation terminates as soon as the complete answer has been found, which may consider the specification of a stop filter.

The LOGA$^+$ can be formulated as the following iterative program :

DELTA := BASIS

```
CLOSURE     :=     Selection_1 {BASIS}
repeat
                   AUX := CLOSURE o DELTA
                   CLOSURE := CLOSURE + AUX
                   DELTA := DELTA o DELTA
until              (No generation of new tuple in CLOSURE) or (DELTA = ∅) or
                   "Stop filter condition satisfied"
CLOSURE     :=     Selection_2 {CLOSURE}
```

In the program, BASIS represents the input relation and CLOSURE the result. At first, DELTA is initialized to BASIS, and CLOSURE is built by selecting its relevant data from BASIS. That is, if the queries require the whole transitive closure, CLOSURE contains all tuples of BASIS; if there are constants in the queries, these are considered by the selection already mentioned. AUX holds the newly generated tuples in each iteration that are then appended to CLOSURE. During the computation, CLOSURE initially holds the intermediate results but at the end holds the complete answer for the query. The process terminates either when CLOSURE stabilizes or DELTA becomes empty or the end filter condition expressed in the query is satisfied.

The result of a whole TC-computation is given stepwise as follows:

After the first iteration:

$$\mathrm{AUX}_1 = \mathrm{CLOSURE}_0 \text{ o } \mathrm{DELTA}_0 = \mathrm{BASIS}^1 \text{ o } \mathrm{BASIS}^1 = \mathrm{BASIS}^2$$

$$\mathrm{CLOSURE}_1 = \mathrm{CLOSURE}_0 + \mathrm{AUX}_1 = \mathrm{BASIS}^1 + \mathrm{BASIS}^2$$

$$\mathrm{DELTA}_1 = \mathrm{DELTA}_0 \text{ o } \mathrm{DELTA}_0 = \mathrm{BASIS} \text{ o } \mathrm{BASIS} = \mathrm{BASIS}^2$$

After the second iteration:

$$\mathrm{AUX}_2 = \mathrm{CLOSURE}_1 \text{ o } \mathrm{DELTA}_1 = (\mathrm{BASIS}^1 + \mathrm{BASIS}^2) \text{ o } \mathrm{BASIS}^2 = \mathrm{BASIS}^3 + \mathrm{BASIS}^4$$

$$\mathrm{CLOSURE}_2 = \mathrm{CLOSURE}_1 + \mathrm{AUX}_2 = \mathrm{BASIS}^1 + \mathrm{BASIS}^2 + \mathrm{BASIS}^3 + \mathrm{BASIS}^4$$

$$\mathrm{DELTA}_2 = \mathrm{DELTA}_1 \text{ o } \mathrm{DELTA}_1 = \mathrm{BASIS}^2 \text{ o } \mathrm{BASIS}^2 = \mathrm{BASIS}^4$$

After the k-th iteration

$$\mathrm{AUX}_k = \sum_{i=2^{k-1}+1}^{2^k} \mathrm{BASIS}^i$$

$$\mathrm{CLOSURE}_k = \sum_{i=1}^{2^k} \mathrm{BASIS}^i$$

$$\mathrm{DELTA}_k = \mathrm{BASIS}^{2^k}$$

So $\lceil \log_2 N \rceil$ iterations for a base relation with recursion depth N are necessary.

In the following we display some characteristics of LOGA^+:

1. LOGA^+ is easily understood and can be effectively implemented by tailored operations in a DB-environment.
2. It presents a very high convergence factor by means of which only $\lceil \log_2 N \rceil$ iterations are necessary for a base relation with recursion depth N, while most of the existing strategies need N or even 2N times.
3. Redundant work is greatly reduced by generating non identical tuples of DELTA in each iteration.

4. Queries with constants are efficiently processed by performing an early selection to pick out relevant data as well as by using query constants as termination conditions to finish the evaluation as soon as possible.
5. Set-oriented queries are also efficiently evaluated by appropriately selecting the relevant data during initialization. It is obvious that $LOGA^+$ is adequate for computing a whole TC as well, if a user wishes it.
6. Termination is guaranteed even in the case of cyclic data by determining the stability of CLOSURE.

Here, we should mention that the proposed algorithm might seem to be similar to the smart algorithm which was described by means of an operator model in [Io86]. However, the smart algorithm has been developed mainly considering the complete computation of a TC, and furthermore, [Io86] makes no explicit statements about the conditions for termination. $LOGA^+$, on the other hand, also aims at the evaluation of queries with constants, presenting an acceptable performance in these cases (See section 4.2).

3.2 The Proof of the Correctness of the Algorithm

In this section, we would like to prove that $LOGA^+$ produces the right and the complete answer for a query independent of the data being processed. For sake of simplicity, we demonstrate this by means of the computation of a whole TC.

Firstly we introduce the notion of "path class" and "distance class" from [He86] (Notice that the symbol "+" is interchangeably used as an arithmetical operation and as DB operation).

Definition:

(1) $W[j] := B^j$ ***(path class)***
This is a relation that contains couples composed of a start node and an end node in the graph having the path length j.

(2) $A[j] := B^j / (B^1+B^2+ \dots B^{j-1})$ ***(distance class)***
This is a relation containing all couples of a start node and an end node in the graph having j as the length of the shortest path between them.

(3) $W[i \dots j] := W[i] + W[i+1] + \dots W[j]$
This is a relation that contains all couples of start and end nodes in a graph having paths with the length L and $i \leq L \leq j$.

(4) $A[i \dots j] := A[i] + A[i+1] + \dots A[j]$
This is a relation that contains all couples of start and end nodes in a graph having the shortest paths with length L and $i \leq L \leq j$.

From the definitions above, properties of the path class and distance class are derived in [He86]. We list two of them that are necessary for our proof:

(5) $(W[a] + W[b]) \text{ o } W[c] = (W[a] \text{ o } W[c]) + (W[b] \text{ o } W[c])$

(6) $W[i] \text{ o } W[j] = W[i+j]$

The following invariant holds during the evaluation:

$$CLOSURE_i = W[1 \dots 2^i],$$

$$DELTA_i = W[2^i]$$

Proof by induction on i:

i = o :

$CLOSURE_o = W[1],$

$DELTA_o = W[1]$

i ---> i + 1

$$
\begin{aligned}
AUX_{i+1} &= CLOSURE_i \text{ o } DELTA_i \\
&= W[1 \ldots 2^i] \text{ o } W[2^i] \\
&= (W[1] + W[2] + \ldots + W[2^i]) \text{ o } W[2^i] && \text{from (3)} \\
&= (W[1] \text{ o } W[2^i]) + (W[2] \text{ o } W[2^i]) + \ldots + (W[2^i] \text{ o } W[2^i]) && \text{from (5)} \\
&= W[2^i+1] + W[2^i+2] + \ldots W[2^{i+1}] && \text{from (6)} \\
&= W[(2^i+1),(2^i+2) \ldots 2^{i+1})] && \text{from (3)} \\
DELTA_{i+1} &= DELTA_i \text{ o } DELTA_i \\
&= W[2^i] \text{ o } W[2^i] \\
&= W[2^{i+1}] && \text{from (6)} \\
CLOSURE_{i+1} &= CLOSURE_i + AUX_{i+1} \\
&= W[1 \ldots 2^i] + W[(2^i+1),(2^i+2) \ldots 2^{i+1}] \\
&= W[1] + W[2] + \ldots W[2^{i+1}] && \text{from (3)} \\
&= W[1 \ldots 2^{i+1}] && \text{from (3)}
\end{aligned}
$$

That is, after the i+1th iteration CLOSURE contains all paths of length L where $L \leq 2^{i+1}$. Finally, CLOSURE holds the transitive closure of the base relation; i.e., all reachable nodes for each node in the base graph.

4. Performance Comparison

In order to show the features of $LOGA^+$ not only theoretically, we have done some performance measurements comparing it with a number of proposed strategies, in order to provide us with a quantitative estimate for the performance of $LOGA^+$ relative to the others. For this comparison, we have chosen the following strategies:

Naive Evaluation (ne) [Ba86],

Semi Naive Evaluation (sne) [Ba86],

Delta Transformation (dt) [Ba85],

Recursive Query/Subquery (rqsq) [Vi86],

Static Filtering (sf) [KL86, BR86],

Generalized Magic Sets (gms) [BMSU86, BR88b],

Generalized Counting (gc) [SZ87, BR88b],

Generalized Supplementary Magic Sets (gmsc) [BR88b] and

Generalized Supplementary Counting (gsc) [BR88b].

Notice that some of the above strategies can also be used for evaluating more complex recursive queries. An analysis of such queries is however beyond the scope of this paper. In this section, we firstly describe our measurements and then give an interpretation of the results. It is important to mention that we have considered specified constants, "moving" them in the evaluation, when computing recursive queries with the Semi Naive and Delta Transformation. Although such consideration cannot be generalized to more complex recursive queries, it is possible for TC-problem, and is, for this reason, significant for our measurement purposes.

4.1 Framework of the Measurements

In our context, we have chosen some typical rules defining a TC-problem and a query with a constant which often occurs in practical application. Hence, we have the following workload:

rules: CLOSURE(X,Y) ← BASIS(X,Y)

CLOSURE(X,Y) ← CLOSURE(X,Z), BASIS(Z,Y)

query: ?CLOSURE(const, X)

In our performance evaluation, we assume that the base relation is binary. A real relation may not be binary, but for most transitive closure queries, only two attributes are involved in the iterative processing. Therefore, it is beneficial to obtain a projection of the relation on these two attributes, before iteratively processing the relation .

As already mentioned in sect. 1, the discussed recursion strategies are primarily used in the context of a DBS in order to extend its functions to that of a DDBS. Therefore we have chosen a DBS for the execution of the various strategies. The DB consisted of a single relation; its size was determined to capture the essential effects of the algorithms running on the different structure types. We didn't aim at measuring the I/O behavior of our algorithms as a function of the problem size. We rather focussed on the investigation of their run time in a DB environment where a large portion (or even all) of the data is kept in the DB buffer to avoid data replacement as much as possible. Since the size of the DB buffer remained fixed for all measurements, we derived comparable results for all algorithms to be processed on a given structure type.

The actual measurements were performed on the commercial DBS INGRES [Da87] in single-user mode under UNIX. The different recursion algorithms were implemented in C with embedded EQUEL-statements (simulating the RA-operators). The DB buffer size was chosen to be 23 pages (default size) where a page of 2048 bytes is the unit of transfer between the DB buffer and external memory.

We have characterized the test data by the following parameters:

N_{edge}: number of the edges in the base graph, equally representing the cardinality of the base relation.

L: the length of the longest path in the base graph, expressing the recursion depth of the base relation.

N_{edge}/N_{node}: ratio of the number of the edges and the number of the nodes, defining in some sense the complexity of the graph.

Structures like list, cycle, full binary tree and random graphs were used to construct the graphs for the measurement. The parameters in Table 1 characterize such structures:

Structure type	Parameters	Remarks
List	List length (5 to 40)	$N_{edge} = N_{node} - 1$, $L = N_{edge}$
Cycle	Cycle length (5 to 40)	$N_{edge} = N_{node}$, $L = N_{edge}$
Full binary tree	tree depth (2 to 10) and number of tuples (6 to 2046)	$N_{edge} = N_{node} -1$ $N_{edge} = 2^{L+1} - 2$
random graph	complexity of a graph N_{edge}/N_{node} (0.6 to 1.4)	each node in the graph has 1/2 N_{edge}/N_{node} entering edges and leaving edges on average.

Tab. 1: Data characterization

To construct the graphs, that is, the tuples with the corresponding relationships, we developed a graph generator that accepts the above parameters as input and randomly creates attribute values with a uniform distribution, according to the specified parameter values. Moreover, isolated nodes in a graph are excluded, but (n:m) - relationships, cyclic and asynchronous data are considered. Hence, the result is a relation managed by INGRES which reflects the structure type unter consideration.

The varying ranges of the parameter values are listed in parentheses in Tab.1, where the number of the tuples in the base relation varied max. to 2046. In the cases of List and Cycle, the size of DB was chosen to be very small, but it can turned out to be sufficient to demonstrate the behavior of the strategies depending on the parameter chosen.

4.2 Results and Interpretation

In this text we focus on the performance behavior of $LOGA^+$ and therefore do not pay so much attention to the other strategies. A detailed analysis of them can be found in [Ya89].

With the first measurement we investigated the dependence of the algorithm’s efficiency on the recursion depth of the data. Therefore, in this case, the test database contained a list with a length varying from 5 to 40 in steps of 8. A constant for the queries was always the head element of the list. Figure 1 shows the results of this experiment. Notice that $LOGA^+$ is not as sensitive to the recursion depth as the other strategies. That is, with an increasing recursion depth $LOGA^+$ shows a better performance behavior in comparison to the other strategies. This is due to it’s convergence factor. Instead of N or even 2N iterations that are necessary in most strategies, $LOGA^+$ needs only $\lceil \log_2 N \rceil$ iterations for TC-computation of a relation having recursion depth N.

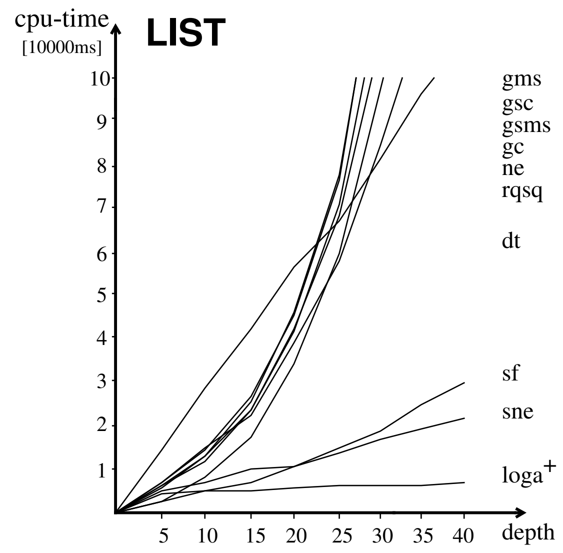

Fig. 1: Comparison of the strategies depending on the recursion depth

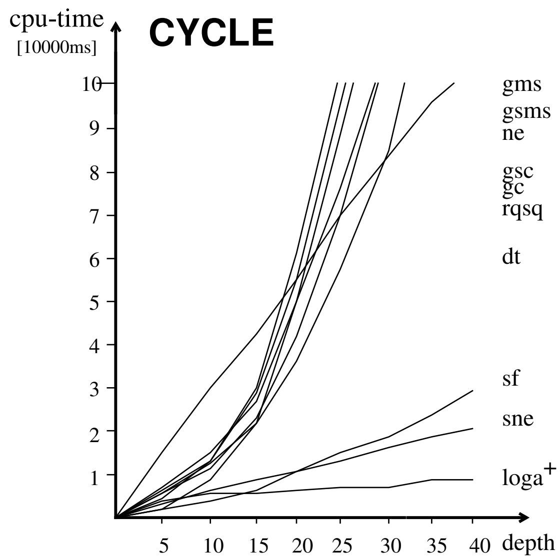

Fig. 2: Comparison of the strategies depending on cyclic data

The second experiment explored, in particular, the behavior of the algorithms when dealing with cyclic data. For this reason, the test database was initialized with a cycle having a variable path length. Except for existence of a cycle, the set of data is similar to the one used in the first experiment. Because of this, figure 2 looks just like figure 1, however with slightly increase of cost caused by a larger number of tuples that are kept in the temporary relations due to the cyclic data.

The next experiment studied the sensitivity of the algorithms to the amount of data being processed. We initialized our test database with a full binary tree, having its height as the recursion depth, because queries were always stated at the root of the tree. Since the number of tuples increase exponentially with the height of the tree, the amount of data plays a key role in this case. Figure 3 shows the result of this experiment (Fig. 3.2 is an enlargement of the shadow part in Fig 3.1). It has been demonstrated that LOGA^{+} still shows a good behavior, however not differing from other strategies (dt, sf, sne) as much as in Fig. 1 and Fig. 2. This is due to the fact that LOGA^{+} accelerates the evaluation by an exponential convergence factor but has to deal, however, with more tuples in each iteration. This is clearly shown in the case of large amounts of data.

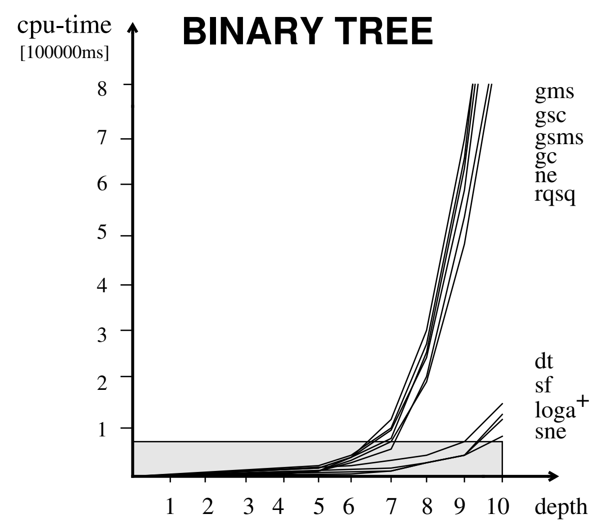

Fig. 3.1: Comparison of the strategies depending on the amount of data

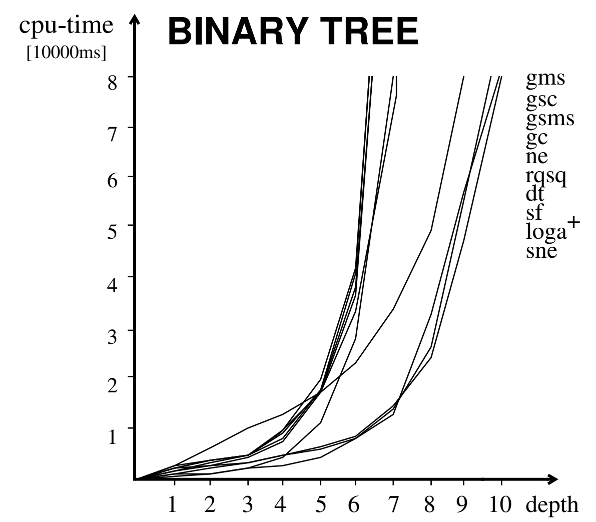

Fig. 3.2: Comparison of the strategies depending on the amount of data

Another experiment studied the case of queries with constants in which the part of relevant data varied. Figure 4 illustrates its result. The test database held a binary tree having a fixed height of 10, but we have stated the queries at different subtrees of the base relation. By doing this, the amount of relevant data varied from 100 % to 0.2 %. We see from Fig. 4 that LOGA$^+$ is not very sensitive to the variation of the amount of relevant data. This is because the computation of Delta = Delta * Delta always considers the entire data of the base relation, in spite of the selection for CLOSURE before the iteration. Notice that we did not measure Naive Evaluation in this experiment, since it computes in all cases the whole TC and then selects the correct answers.

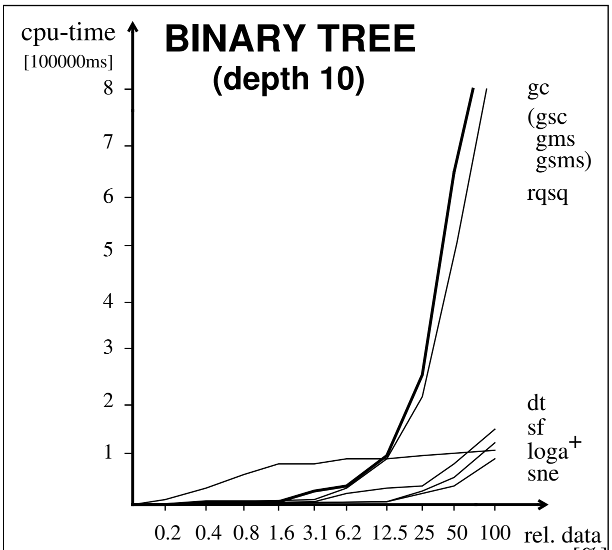

Fig. 4: Comparison of the strategies depending on the relevant data

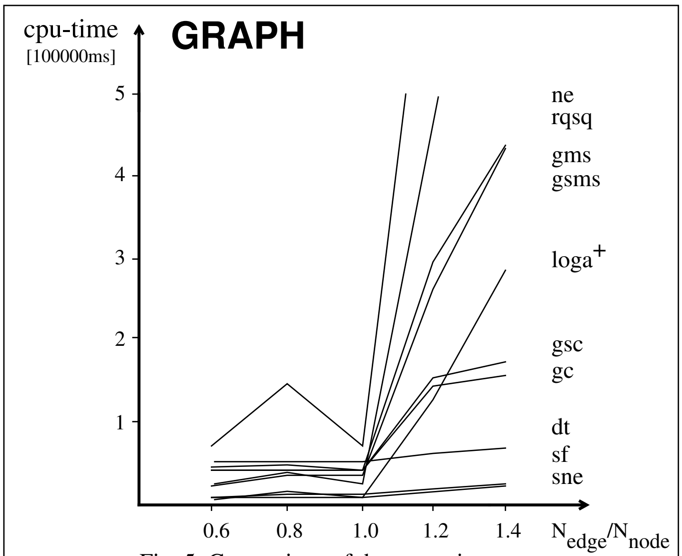

Fig. 5: Comparison of the strategies depending on the complexity of graphs

The last experiment gives a comparison of the algorithms in terms of the connectivity of the data. The test database was loaded with randomly generated graphs having a variable relationship between edges and nodes by keeping the number of edges fixed, and the recursion depth of the relation equal to 10 (figure 5). We see that LOGA$^+$ is relatively sensitive to the complexity of a graph. After the relationship N_{edge}/N_{node} is greater than 1, the cost of LOGA$^+$ is not very low.

Our emphasis was on a uniform environment for all strategies to be compared. Therefore the relative performance difference in Fig. 1 to Fig. 5 are indicative for the quality of the different strategies. An evaluation of their "absolute" performance behavior would have required the consideration of much more parameters (e.g. different DB buffer sizes) which was beyond the scope of our study.

Finally to be noticed is that the group of the optimization strategies including gms, gc, gsms and gsc gives worse performance behavior in some cases than Naive Evaluation. These strategies try to optimize by means of a search for the relevant data using the specified query constant and then compute the whole TC of the relevant data. In our experiments (Fig. 1, 2 and 3) all data is relevant to the query, thus a search for relevant data brings nothing but a rise in cost. This is further discussed in [Ya89].

5. Conclusion and Outlook

Transitive closure operation has been widely recognized as a necessary extension to relational query languages. In this paper, we have discussed some important aspects to be considered when developing an algorithm for the computation of the transitive closure of a relation. These are referred to as the fast convergence, the prevention of redundant work, processing of queries with constants including set-oriented ones, and handling of cyclic data. Following, we presented the LOGA$^+$ algorithm for TC-computation and the results of a set of measurements, comparing it with some other strategies. LOGA$^+$ is an improvement on Naive Evaluation that can be effectively used as an algorithm for TC computation. It shows good performance, particularly for data with large recursion depth. Certainly, some further improvements are necessary in order to make it also very efficient in processing data with a high degree of interconnection.

From a practical point of view, many recursive queries refer not only to pure reachability computation but also to some additional information e.g. the shortest paths in a graph. Such queries are called generalized transitive closure [DB86], and their processing deserves extra attention on some issues, e.g. loss of path information by merging immediate results of computation. Solving such kind of problems is also a topic that deserves some further research.

Acknowledgment

We would like to thank Prof. T. Härder for his helpful comments which contributed to clarify and improve the ideas presented in this paper. Also acknowledged are the careful reading and useful hints of S. Deßloch as well as the measurements performed by N. Ritter. Finally, we would like to express our thanks to the anonymous referees for their helpful comments on the paper.

References

[AJ87] Agrawal, R., Jagadisch, H.: Direct Algorithms for Computing the Transitive Closure of Database Relations, in: Proc. of the 13th VLDB Conference, Brighton 1987, pp. 255-266.

[Ba85] Bayer, R.: Database Technology for Expert Systems, in: Proc. GI-Conference on Knowledge-Based Systems, Informatik-Fachbereichte 112, Springer-Verlag, Munich, Oct. 1985, pp. 1-16.

[Ba86] Bancilhon, F.: Naive Evaluation of Recursively Defined Relations, in: On Knowledge Base Management Systems - Integrating Database and AI Systems, (Ed: Brodie and Mylopoulos), Springer-Verlag, 1986, pp. 165-178.

[BMSU86] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic Sets and Other Strange Ways to Implement Logic Programs, in: Proc. of the Fifth ACM SIGACT-SIGMOD Symposium on Prinziples of Database Systems, Cambridge, Massachusetts, March,1986.

[BR86] Bancilhon, F., Ramakrishnan, R.: An Amateur's Introduction to Recursive Query Processing Strategies, in: Proc. of SIGMOD '86, Inter. Conf. on Management of Data, SIGMOD Record, Vol. 15, No. 2, June 1986, pp. 16-52.

[BR88a] Bancilhon, F., Ramakrishnan, R.: Perfoemance Evaluation of Data Intensive Logic Programs, in: "Foundations of Deductive Databases and Logic Programming" (Ed: Minker, J.), Morgan Kaufmann Publishers, 1988, pp. 439-511.

[BR88b] Beeri, C., Ramakrishnan, R.: On the Power of Magic, Computer Sciences Technical Report #770, University of Wisconsin-Madison, to appear in the Journal of Logic Programming.

[Da87] Date, C.J.: A Guide to INGRES, Addison-Wesley Publishing Company,1987.

[DS86] Dayal, U., Smith, J.: PROBE: A Knowledge-Oriented Database, in: On Knowlege Base Management Systems (Ed: Brodie and Mylopoulos), Springer-Verlag, 1986, pp. 227-257.

[GMN84] Gallaire, H., Minker, J., Nicolas, J.M.: Logic and Databases: A Deductive Approach, in: ACM Computing Surveys, Vol. 16, No. 2, June 1984, pp. 153-186.

[He86] Heigert, J.: Efficient Algorithms for Relational Database Systems Supporting Recursion (in German), TUM-18613, TU Munich, July 1986.

[HL86] Han, J., Lu, H.: Some Performance Results on Recursive Query Processing in Relational Database Systems, in: IEEE Proc. of Inter. Conf. on Data Engineering, Los Angeles, Feb. 1986.

[HQC88] Han, J., Qadah, G., Chaou, C.: The Processing and Evaluation of Transitive Closure Queries, iin: Advances in Database Technology - EDBT '88, Inter. Conf. on Extending Database Technology, Venice, Italy, March 1988, Sringer-Verlag, pp. 49-75.

[Io86] Ioannidis, Y.: On the Computation of the Transitive Closure of Relational Operators, in: Proc. of the 12th VLDB Conference, Kyoto, 1986, pp. 403-411.

[IW88] Ioannidis, Y., Wong, E.: Transforming Nonlinear Recursion to Linear Recursion, in: Proc. of the Second Inter. Conf. on Expert Database Systems, EDBS '88, Tysons Corner, Virginia, Apr. 1988.

[JAN87] Jagadish, H., Agvawal, R., Ness, L.: A Study of Transitive Closure as a Recursion Mechanism, in: Proc. of ACM SIGMOD 1987 Annual Conference, San Francisco, May 1987, pp. 331-344.

[KL86] Kifer, M., Lozinskii, E.: A Framework for an Efficient Implementation of Deductive Databases, in: Proc. of Advanced Database Symposium, Tokyo, 1986.

[LMR87] Lu, H., Mikkilineni, K., Richadson, J.: Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation, in: Proc. of the Third Inter. Conf. on Data Engineering, Los Angeles, Feb. 1987, pp. 112-119.

[SZ87] Sacca, D., Zaniolo, C.: Magic Counting Methods, in: Proc. of ACM SIGMOD 1987 Annual Conference, San Francisco, May 1987, pp. 49-59.

[Vi86] Vieille, L.: Recursive Axioms in Deductive Databases: the Query/Subquery Approach, in: Proc. of First Inter. Conf. on Expert Database Systems, Charleston, 1986, pp. 179-193.

[Ya87] Yan, W.: Analysis of Different Strategies for the Processing Recursive Queries in Deductive Database Systems (in German), Technical report, University of Kaiserslautern, Nov. 1987.

[Ya89] Yan, W.: An Overview of Existing Strategies for the Evaluation of Recursive Queries (in German), Internal Report, University of Kaiserslautern, in preparation.

[ZY87] Zhang, W., Yu, C.: A Necessary Condition for a Doubly Recursive Rule to be Equivalent to a Linear Recursive Rule, in: Proc. of ACM SIGMOD 1987 Annual Conference, San Francisco, May 1987, pp. 345-356.