An Overview of Parallel Strategies for Transitive Closure on Algebraic Machines *

Filippo Cacace
Dipartimento di Elettronica, Politecnico di Milano

Stefano Ceri
Dipartimento di Matematica, Universita' di Modena

Maurice A.W. Houtsma
Department of Applied Mathematics, University of Twente

Abstract

An important feature of database technology of the nineties is the use of distributed computation for speeding up the execution of complex queries. Today, the use of parallelism is tested in several experimental database architectures and a few commercial systems for conventional select-project-join queries. In particular, hash-based fragmentation is used to distribute data to disks under the control of different processors, in multi-processor architectures without shared memory, in order to perform selections and joins in parallel.

With the development of new (logic) query languages and deductive databases, the new dimension of recursion has been added to query processing. Transitive closure queries, such as bill-of-material, allow important database problems to be solved by the database system itself; and more general logic programming queries allow us to study queries not considered before. Although recursive queries are very complex, their regular structure makes them particularly suited for parallel execution. Well-considered use of parallelism can give a high efficiency gain when processing recursive queries.

In this paper, we give an overview of approaches to parallel execution of recursive queries as they have been presented in recent literature. After showing that the most typical Datalog queries have exactly the same expressive power as the transitive closure of simple algebraic expressions, we focus on describing algebraic approaches to recursion.

To give a good overview of the problems that are inherent to parallel computation, we introduce a graphical formalism to describe parallel execution. This formalism enables us to clearly show the behaviour of parallel execution strategies. We first review algorithms developed in the framework of algebraic transitive closures that operate on entire relations; then we introduce fragmentation, distinguishing between hash-based and semantic fragmentation.

1 Introduction

Over the past few years, recursive queries have emerged as an important new class of complex queries. These queries enable solving classical database problems, such as the *bill-of-material* (finding all transitive components of a given part). This type of problems is classically managed in commercial applications by embedding queries within programming language interfaces, and then dealing with recursion by using the programming language constructs. However, these applications are both

*This research is partially supported by the LOGIDATA+ Project of the National Research Council of Italy unefficient and hard to program. Database languages of the future will be able to express simple types of recursion, such as transitive closure, within their query languages; and logic programming interfaces to databases will be able to express general recursion. These queries are intrinsically much more complex than conventional queries, because they require iterating the application of operations until termination (fixpoint) conditions are met. Though in general the finitedness of the computation is certain, the number of iterations is not known a-priori. Thus, there is a definite need for the development of new optimization strategies and techniques to deal efficiently with recursive queries.

During the past decade, much effort has been put into the development of new techniques for more efficient processing of conventional relational queries. These techniques range from the use of efficient physical data structures, to a clear separation between *clients* and *servers*, and to the use of multi-tasking and multi-threading within advanced architectures for database servers. Although these techniques can be supported in a conventional single-processor environment, they are particularly suited for multi-processor architectures. These architectures are becoming more and more widespread, be it in the context of distributed systems or in the context of advanced multi-processor machines. Indeed, database access is particularly suited for distributed (parallel) execution, because it adds the dimension of *data distribution* to processing distribution, thus allowing a combination of these two dimensions to generate very efficient execution strategies.

We may distinguish two types of parallelism in databases. *Inter-query parallelism* enables multiple small queries to be executed in parallel; this notion of parallelism is used for building systems capable of running hundreds or even thousands of small transactions per second against a large, shared database. In this case, efficiency is mainly achieved by building servers that are very fast in processing each request; parallelism is the consequence of the concurrent presentation of requests from multiple sources, which are served concurrently by a complex process architecture; the database itself may or may not be distributed. In the rest of the paper, we will not consider this type of parallelism.

In this paper we concentrate on the second type of parallelism, called *intra-query parallelism*, which enables the distribution of complex queries to multiple processors. Intra-query parallelism aims at minimizing the response time of a query by sharing the heavy processing load on multiple processors, in this case each processor is typically dedicated to the query.

Intra-query parallelism has been considered as an important feature in query optimization since many years. Relational query optimizers try to exploit parallelism by detecting the parts of a query plan that can be executed in parallel; asynchronous models of execution are typically used even within a centralized database architecture in order to enable the concurrent execution of parts of an access plan. In distributed databases, intra-query parallelism has been considered as an underlying, implicit assumption of many theoretical approaches to query optimization developed in the late seventies and early eighties, which were building fast execution plans by postulating that each part of the plan could be executed in parallel [2, 16].

Intra-query parallelism has become feasible only very recently. And although many systems support parallelism in query execution, they do not yet support fragmentation; only few commercial systems support fragmentation, including Teradata and Tandem [29]. Advanced techniques based on data fragmentation and performance measurements for comparing them have been developed in a few research environments; among them, the *Prisma* machine, developed at Philips and several Dutch universities [4, 24], the *Delta* machine, developed at Wisconsin University [14, 28], and the *Bubba* machine, developed at MCC [13].

Especially data fragmentation is an essential ingredient for parallelism, as it enables a natural partitioning of query processing. Each processor controls a disk which stores fragments of relations. With this architecture, it is possible to execute selections, projections, and some joins in a distributed way on each processor, and collect from each processor the result of these operations [10]. Due to the importance of fragmentation for intra-query parallelism, we will in our overview of parallel strategies separate methods operating on fragments from those operating on complete relations.

In this paper, we survey parallel techniques for executing recursive queries on algebraic machines. Section 2 presents some preliminary foundations, and in particular builds the bridge between research on transitive closure for algebraic expressions and research on Datalog and logic programming. We show that these two research fields, though apparently different, are indeed two sides of the same coin. After having shown this, we may freely mix logic and algebraic formalisms for expressing recursive queries and evaluation strategies. We also introduce a graphical representation of algorithms, highlighting regularities in the computation and clearly showing the parallel behaviour of the evaluation strategies to be discussed.

In Section 3, we describe research focused on parallel execution of transitive closure. First, we discuss methods which do not use fragmentation, then we discuss hash-based fragmentation, and finally we discuss fragmentation that is based on the semantic content of data.

2 Preliminary foundations

When we consider research for the optimization of recursive queries, we note that the problem is approached from two different perspectives: algebraic optimization and logic optimization. Examples of research on algebraic optimization may be found in [3, 9, 12, 18, 31], examples of logic optimization may be found in [15, 22, 26, 32, 33]. These two approaches are actually equivalent; this is illustrated in [7, 8], where it is described how to transform Datalog programs into equations of positive Relational Algebra (RA+), and *vice versa.*

Let us show this equivalence in detail. The transitive closure of a binary relation R is expressed by the following formula, for some finite number n:

$$R^* = R \cup R^2 \cup R^3 \cup \ldots \cup R^n$$

Note that in this formula $R^2 = \pi_{1,4}(R \underset{2=1}{\bowtie} R)$, $R^3 = \pi_{1,4}(R^2 \underset{2=1}{\bowtie} R)$, etc. The solution of this expression is equivalent to the minimal model of the following Datalog program (where lower case letters denote Datalog predicates, relations corresponding to these predicates are denoted by upper case letters):

$$\begin{aligned} p(X,Y) &:- r(X,Y). \\ p(X,Y) &:- r(X,Z),\ p(Z,Y). \end{aligned}$$

Of course, this also holds in the other direction. Consider the following Datalog program:

$$\begin{aligned} p(X,Y) &:- r(X,Y). \\ p(X,Y) &:- s(X,S),\ t(S,T),\ p(T,Y). \end{aligned}$$

This program consists of one linear stable recursive rule and one non-recursive rule; it is called a *linear sirup* in logic programming terminology. The minimal model of this program can be computed by unfolding the recursive rule until no more new tuples are derived. Hence, in the first step we get:

$$p(X,Y) : -s(X,S), t(S,T), r(T,Y)$$

which is equivalent to:

$$\pi_{1,4}(\pi_{1,4}(S \underset{2=1}{\bowtie} T) \underset{2=1}{\bowtie} R)$$

In the second step we get:

$$p(X,Y) : -s(X,S), t(S,T), s(T,U), t(U,V), r(V,Y)$$

which is equivalent to

$$\pi_{1,4}(\pi_{1,4}(\pi_{1,4}(S \underset{2=1}{\bowtie} T) \underset{2=1}{\bowtie} \pi_{1,4}(S \underset{2=1}{\bowtie} T)) \underset{2=1}{\bowtie} R) = \pi_{1,4}((\pi_{1,4}(S \underset{2=1}{\bowtie} T))^2 \underset{2=1}{\bowtie} R)$$

and so on. By unfolding the definition in this way and translating it to Relational Algebra, we compute $(\pi_{1,4}(\pi_{1,4}(S \underset{2=1}{\bowtie} T))^* \underset{2=1}{\bowtie} R)$; this is an ordinary transitive closure.

Summarizing, the following proposition holds:

```
power := $R$;
union := $R$;
**repeat**
        old_union := union;
        power := $\pi$ (power $\bowtie$ $R$);
        union := union $\cup$ power
**until** union = old_union

delta := $R$;
union := $R$;
**repeat**
        power := $\pi$ (delta $\bowtie$ $R$);
        delta := power - union;
        union := union $\cup$ delta
**until** delta = $\emptyset$
```

Figure 1: Naive and semi-naive algorithm

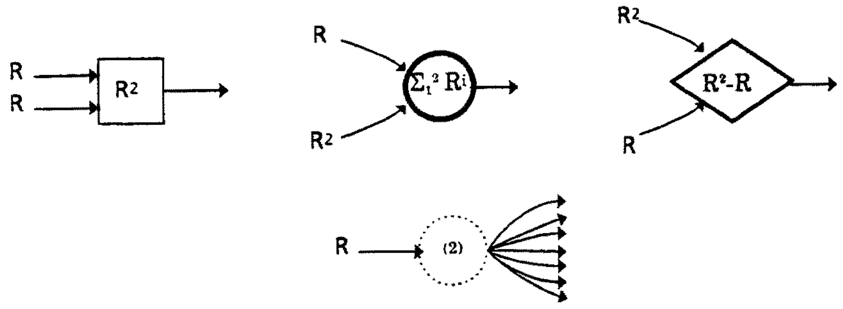

Figure 2: Building blocks of the graphical model

- Each linear sirup in Datalog is equivalent to a transitive closure operation.

Since transitive closure and linear sirups in Datalog are equivalent, we may use the same procedure for computing them. The two most well-know methods are called *naive* and *semi-naive*; they are shown in Fig. 1. Their relative merit is discussed, for instance, in [8]. Briefly, semi-naive evaluation is more efficient than naive evaluation, because at each iteration only the difference term *delta* is joined, instead of the entire term *power*. Semi-naive evaluation can be applied to linear Datalog programs (and therefore to sirups) but it cannot be applied to general Datalog programs.

Knowing that Datalog and positive Relational Algebra may be easily translated into each other, in the rest of this paper we freely mix the two formalisms; in general, we use Datalog to express queries and Relational Algebra for describing evaluations. To improve and clarify the description of algorithms, we introduce a graphical model to represent algebraic expressions; the graphic representation highlights the structure of parallel computation.

The building blocks of the graphical model are graphic symbols, each corresponding to either an algebraic operation or to a hashing operation. Input operands for the operations are represented by input edges, while the result produced by the operation is represented by an output edge. The label inside the building block indicates the operation performed. The following graphic symbols, as shown in Fig. 2, are used:

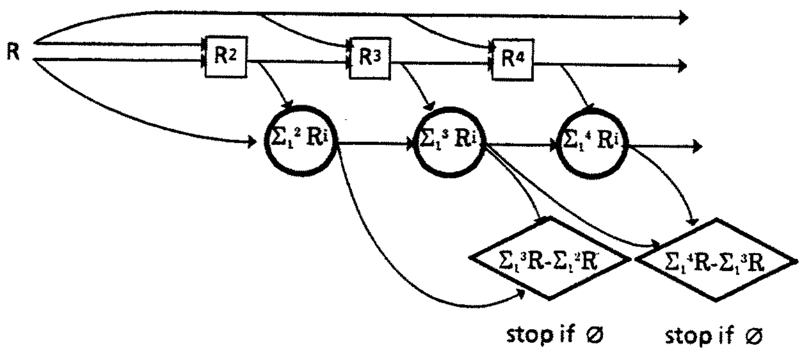

Figure 3: Graphic representation of naive evaluation

Squares are used to denote *joins*. Each join has two operands and produces one result. In Fig. 2 we see a join having a relation R as both inputs and producing R^2 as result. If R represents connections (of length 1), then R^2 represents paths of length 2.

Circles are used to denote *unions*. Each union has several operands and produces one result. In Fig. 2 we see the union $R \cup R^2$, giving all all paths of either length 1 or length 2.

Diamonds are used to denote *set differences*. Each difference has two operands and produces one result. In Fig. 2 we see the set difference $R^2 - R$, giving all paths of length 2 that do not already appear as direct connections.

Dashed circles are used to denote data redistribution by *hashing*. Each hashing operation has one input operand and produces an output operand. The label inside the block indicates the hashing criterion; in Fig. 2 we see that the relation R is hashed on its second attribute.

Fig. 3 presents the naive evaluation of transitive closure queries which was described in an algorithmic way in Fig. 1. The figure clearly shows that three types of operations are involved: joins to compute subsequent powers of R, unions to gather the results, and set differences to test for termination of the computation.

3 Parallel algorithms for transitive closure

In this section we first discuss several variations to naive evaluation that were not designed for parallel evaluation, but are nevertheless instructive to analyse. We then discuss parallel naive evaluation, hash-based fragmentation, and finally fragmentation based on semantic content of data.

3.1 Variations to naive evaluation

Throughout this subsection, we consider variations to the naive evaluation which try to improve its performance by either reducing the number of tuples considered at each iteration (through the semi-naive approach) or by generating several powers of R at each iteration, thus reducing the total number of required iterations (through the square, smart, and "minimal" approach). These algorithms were not designed for parallel evaluation, but their analysis is instructive because it indicates intrinsic limits to parallelism, thus enabling us to identify the features that lead to an inherently sequential behaviour and preventing parallel evaluation.

Algorithms evaluate the extension of the predicate p defined as follows:

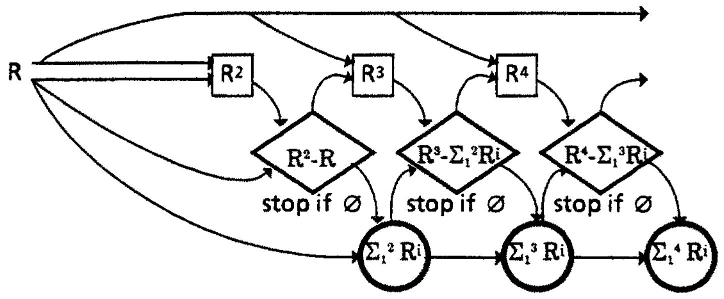

Figure 4: Semi-naive evaluation

$p(X,Y)\ :-\ r(X,Y).$
$p(X,Y)\ :-\ r(X,Z),\ p(Z,Y).$

Such an extension is given by all the tuples belonging to the subsequent powers R^n of the relation R, as discussed in Section 2.

One way of introducing parallelism in the evaluation consists in assigning a specialized processor to each *type of operation*. For instance, one processor performs joins, a second one performs unions, a third one performs differences. In this case, there is a strict sequentialization between operations of the same kind, but operations of different kinds can sometimes occur in parallel. Typically, they are done in parallel if they use the same input relations.

The second approach to parallelism (a brute-force one) consists in assigning a new processor to the evaluation of *each iteration*, until processors are exhausted; in this case, it is essential to use pipelining, so that each processor starts as soon as possible, namely, when its first input tuples are produced by the predecessor processors. Note that set difference is peculiar, because it cannot produce output tuples until its second operand is complete, thus breaking the flow of pipelining. When set differences are only used for the termination test, it may be convenient to perform them asynchronously, so that processes are not slowed down. However, this approach incurs the risk of replicated or superfluous computation, because iterations may be activated also subsequent to a successful termination test: the test becomes known after processing has already occurred. This may cause a dramatic increase of the global amount of work performed, while the reduction of delay might be minimal.

3.1.1 Naive evaluation

Let us reconsider Fig. 3. Unions, joins, and differences may be assigned to a different processor or to a different class of processors. In the naive approach, many duplicate tuples may be generated; this may lead to a high overload of tuples sent across the network. Since set difference determines the termination of the computation, it acts as a synchronization point. Join processors may be slowed until difference is terminated, or instead they can proceed at the risk of performing unnecessary computation. In conclusion, naive evaluation is not very well suited for parallel computation.

3.1.2 Semi-naive evaluation

Fig. 4 shows the semi-naive evaluation of transitive closure queries, which was described before in an algorithmic way in Fig. 1. The graphical presentation clearly shows that also semi-naive algorithm is hardly suited for parallel computation. At each step, a set difference operation is required before the join operation; this operation eliminates duplicates from the computation. Therefore, join and set difference operations are strongly synchronized. The union that builds the final results may

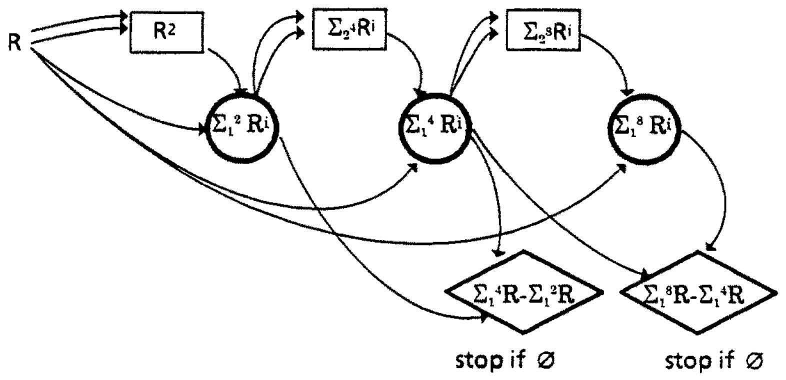

Figure 5: Squaring evaluation

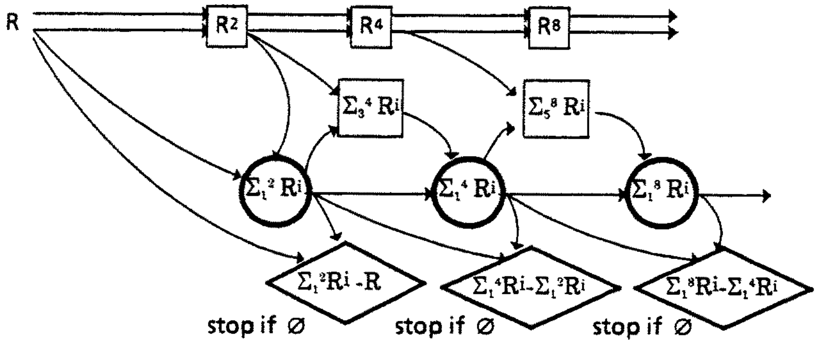

Figure 6: "Smart" evaluation

be done asynchronously on a separate processor, but this does not lead to major improvements in performance. In summary, due to the synchronization imposed by the set difference, the resulting semi-naive evaluation is essentially a sequential process.

3.1.3 Squaring evaluation

Squaring evaluation, introduced by Apers et al. [3] (and others), is shown in Fig. 5. This method reduces the number of iterations required to compute the transitive closure by subsequently squaring the result from the previous iteration. Hence, first paths of length up to 2 are computed, then paths of length up to 4, then paths of length up to 8, and so on. The graphical representation clearly shows that joins and unions are interleaved, while set differences are only performed for testing termination, and can be done asynchronously.

3.1.4 Smart evaluation

"Smart" evaluation, introduced by Ioannidis [23], uses an improved variation of the squaring approach, by considering at each iteration the paths of length 2, 4, 8, etc. to create new tuples. The computation of the subsequent powers of R (R^2, R^4, R^8, etc.) may be done on a separate processor or processor class without synchronizing with the rest of the computation, but unions and other joins

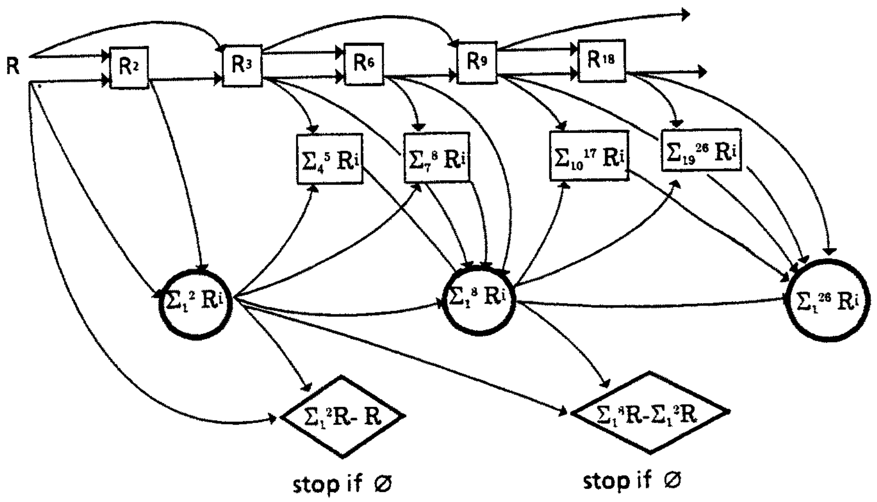

Figure 7: "Minimal" evaluation

required for producing the result must be synchronized with the power joins. Set differences are only used for testing termination and they can be done asynchronously.

3.1.5 Minimal evaluation

Besides the "smart" evaluation, some other rewritings of the transitive closure are also introduced by Ioannidis [23]. One of them is the so-called "minimal" algorithm, shown in Fig. 7. It is called minimal because it requires the minimal number of operations for computation of the transitive closure. Obviously, these operations are in general very complex, so the gain in number of operations is outbalanced by their complexity. Again, joins computing the powers of R do not need to synchronize with other operations, while the other join and union need to. The set difference operation is used only for testing termination.

3.1.6 Concluding remarks

In this section we have shown that standard algorithms for computation of the transitive closure operation are very hard to parallelize. These algorithms were defined in order to compute the entire transitive closure; they can also be used for queries having the first argument of the predicate p bound to a constant, e.g., $? - \; p(a,X).$ or $? - \; p(X,a)$; let R_{bf} and R_{fb} denote the results of these queries.

The straightforward approach for solving these queries is to apply a final selection on R^*; however, the most efficient method consists in anticipating the selection. The method of *variable reduction* [8] allows to simplify the query $? - \; p(X,a)$ on the program:

$$\begin{aligned} p(X,Y) &: - \; r_1(X,Y). \\ p(X,Y) &: - \; r_2(X,Z), \; p(Z,Y). \end{aligned}$$

As it follows:

$$R_{fb} \;=\; \sigma_{2=a}R^k \;= \pi_{14}(R^{k-1} \underset{2=1}{\bowtie} \sigma_{2=a}R)$$

Where k is the iteration number at which the fixpoint is reached, i.e. $R^* = R^k$. This formula, evaluated from right to left, allows computing the first join by using $\sigma_{2=a}R$ as an operand, instead of R. This corresponds to the conventional anticipation of selections wrt. joins in query processing practice.

The query ?$-$ $p(a, X)$. cannot be simplified in the same way; however, if we consider the equivalent program:

$$\begin{aligned} p(X,Y) &:- r_1(X,Y). \\ p(X,Y) &:- p(X,Z),\ r_2(Z,Y). \end{aligned}$$

Then we have:

$$R_{bf} = \sigma_{1=a}R^k = \pi_{14}(\sigma_{1=a}R \underset{2=1}{\bowtie} R^{k-1})$$

This formula, evaluated from left to right, allows computing the first join by using $\sigma_{1=a}R$ as an operand, instead of R.

Performance comparisons between the algorithms discussed in this subsection and the semi-naive method, presented in [23], show in general better performances; however, they use only relations corresponding to trees and list, whereas graphs would be subject to the severe problem of dealing with duplicate tuples in the computation. Thus, real gains of algorithms square, smart, and minimal with cyclic databases are questionable.

3.2 Parallel naive evaluation

In this section we discuss a parallel version of naive evaluation as described by Raschid and Su [27]. They consider again the recursive program:

$$\begin{aligned} p(X,Y) &:- r(X,Y). \\ p(X,Y) &:- r(X,Z),\ p(Z,Y). \end{aligned}$$

They concentrate on the query ? $-$ $r(a,b)$., characterized by having both arguments bound to a constant; let R_{bb} denote the relation produced as result. Note that, for a particular choice a and b of bindings, R_{bb} returns the tuple (a,b) if $r(a,b)$ can be proved, the empty relation otherwise. We further denote as R^i_{bb} the tuples produced at the $i-th$ iteration for solving the R_{bb} query, and similarly for R^i_{bf} and R^i_{fb}; finally, the summation: $\Sigma^k_1 R^i_{bb}$ gives all the result tuples for the query R_{bb} after k iterations.

Fig. 8 shows the algorithm of Raschid and Su [27]. Though in principle this architecture is capable of solving the R_{bf} and R_{fb} queries, in practice the architecture degenerates to the naive evaluation in those cases (assume the R_{bf} query and the first line of joins, unions, and differences for R_{bf} terms with the naive evaluation shown in Fig. 3; the other blocks of this architecture do not produce useful tuples). Thus, this architecture applies successfully only to the R_{bb} query.

Based on the bindings in the query, they start by evaluating R_{bf}, R_{fb} and R_{bb}: we have $R_{bf} = \sigma_{1=a}R$, $R_{fb} = \sigma_{2=b}R$, $R_{bb} = \sigma_{1=a,2=b}R$. Then, at each iteration four terms are computed through join operations: R^i_{bf}, R^i_{fb}, R^{2i-1}_{bb}, and R^{2i}_{bb} (the last term is omitted at the first iteration). In the figure, the four terms are shown for the first three iterations (note a 3×4 matrix of joins); the reader can thus perceive the regularity of the computation.

At each iteration, three union operations are also required: $\bigcup_i R^i_{bf}$, $\bigcup_i R^i_{fb}$, and $\bigcup_i R^i_{bb}$. Additionally, two differences compute the increment of tuples solving the queries R_{bf} and R_{fb}. The query R_{bb} is solved when $\bigcup_i R^i_{bb}$ produces one tuple. It is also solved negatively, in the sense that the answer is the empty relation, when either of the differences is empty; that means that no additional tuples are or will be produced that were not considered at the previous iteration.

When we consider the number of processors to be used in this evaluation mechanism, we may note the following. The four join operations should be done in parallel, and similarly the three unions. The set difference operations are best implemented on the same processors as the corresponding unions. Therefore, the suggested optimal number of processor is seven.

Although parallelism is clearly achieved by this method, some critical remarks should be made. First, the considered query is rather peculiar (R_{bb}); since the computation may be halted after the

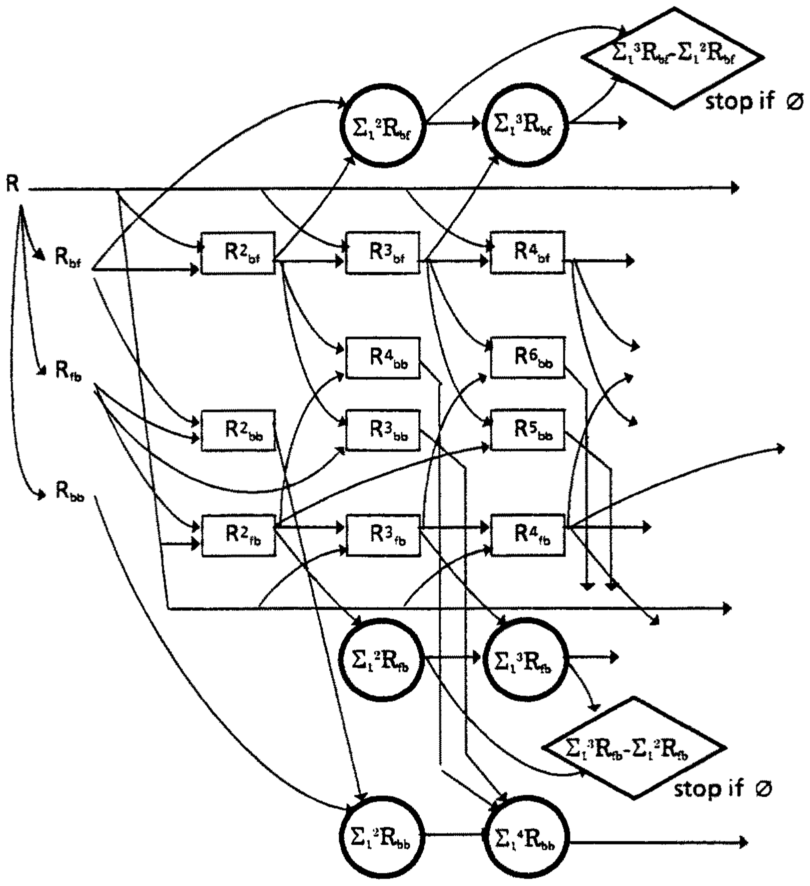

Figure 8: The method by Raschid and Su

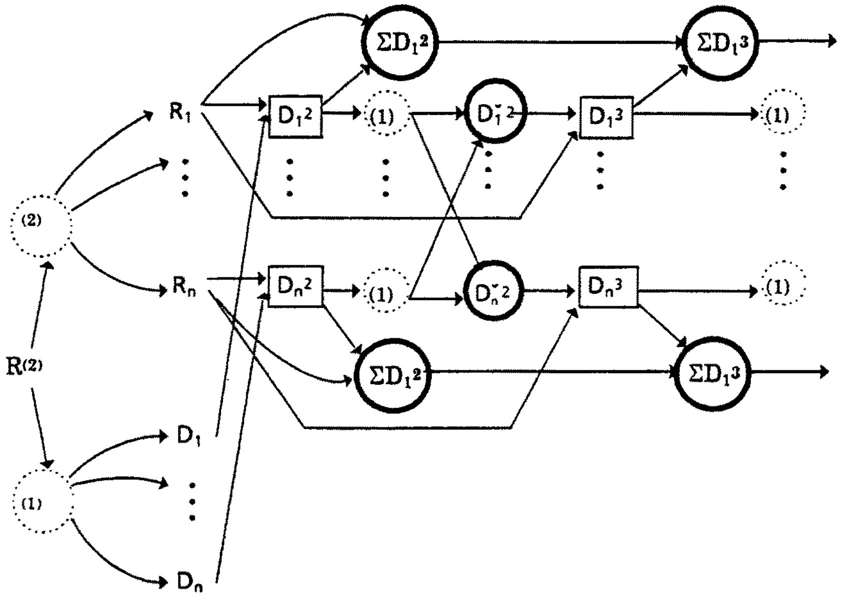

Figure 9: The method by Valduriez and Khosafian

first tuple that answers the query is produced, this approach is rather like cracking a nut with a sledgehammer. Second, parallelism comes together with massive interprocessor communication, and this is rather costly. In [17, 18, 25] it was reported that simulations of this method on a model of the *Prisma* database machine actually showed a negative speed-up, due to synchronization and communication costs. This parallel strategy turned out to be slower than smart single-processor strategies.

3.3 Hash-based fragmentation

In this section we study methods for parallel execution of recursive queries which use hash-based fragmentation. The basic idea behind these strategies is to use fragmentation of relations R and S in order to compute $R \bowtie S$ as $R_1 \bowtie S_1 \cup \ldots \cup R_n \bowtie S_n$. This approach is called *simple distributed join* in [10], where applicability and correctness conditions are discussed.

We consider a fragmentation of R into $R_1, \ldots R_n$ and the iterative join of R with itself in order to solve the usual recursive problem:

$$p(X,Y) \ :- \ r(X,Y).$$
$$p(X,Y) \ :- \ r(X,Z),\ p(Z,Y).$$

We start by discussing the approach of Valduriez and Khosafian [31], then we discuss a straightforward extension of this approach described by Cheiney and de Maindreville [12].

3.3.1 The method by Valduriez and Khosafian

The method by Valduriez and Khoskafian [31] is shown in Fig. 9. The strategy starts by hashing the relation R on its *second* attribute and distributing it to n processors. A second copy of R, called

D, is then hashed on its *first* attribute and distributed to processors; this copy represents the *delta* relation.

This fragmentation has to be fully understood: in practice, the domain *dom* of the join columns of R is partitioned into subdomains dom_i, and each domain is assigned to a processor; then, that processor receives the fragments R_i of R and D_i of D corresponding to that subdomain dom_i. In this way, the join between the second and first column of R can take place in parallel on each processor. However, fragments might be unbalanced; there is no guarantee that, by partitioning the domain and then by building the fragmentation, fragments will be of the same size.

On each processor i, the fragments of R_i and D_i are joined, generating the results: $D_i^2 := R \underset{2=1}{\bowtie} D_i$. This result has to be re-hashed on the first column. Re-hashing is done locally on each processor, and the results are sent to the appropriate processor for the next join ($D_i^3 := R_i \underset{2=1}{\bowtie} D_i^2$). At each step, deltas are accumulated at each processor by means of a union. Note that in this strategy the same tuple may appear in several deltas, thus leading to unnecessary, redundant computations.

When R corresponds to a directed acyclic graph (DAG), the computation ends when all deltas are empty. When instead R corresponds to a relation with cycles, then the union of deltas produced after each iteration at all processors and a set difference with the union produced at the previous iteration are required for detecting termination. In Fig. 9 we only show the accumulation of deltas at each processor; a last step (not shown in Fig. 9) is to gather all accumulated deltas on a single processor by a final union operation.

3.3.2 The method by Cheiney and de Maindreville

In [12], Cheiney and de Maindreville show a simple extension of this evaluation strategy that avoids rehashing deltas at each iteration step. This strategy is shown in Fig. 10; it differs from the approach described in [31] only for one feature: after the hashing of R_i on the second attribute, the resulting n fragments are further hashed on their first attribute. Each of thè n fragments R_i is thus conceptually divided into n sub-fragments R_{ij}. After joining with the delta fragments at each iteration k, the second hashing is used to pre-determine where tuples of delta relations D_i^k need to be sent, without need for their rehashing. Figure 10 shows this algorithm.

Note that this approach can be even further extended by assigning a processor to each sub-fragment R_{ij} instead of assigning a processor to each fragment $R_i = \bigcup_j R_{ij}$. In this way, n^2 processors are used instead of n, each performing a smaller fraction of work.

3.3.3 Concluding remarks

Approaches presented in this section extend to recursive query processing the work currently being done for applying ***intra-query parallelism*** to joins [14, 13, 29]. As we noted in the introduction, recursive queries present a repeating pattern of join operations, hence parallelism has great potential.

The effectiveness of the evaluation strategies presented in this section depends crucially on an even distribution of the workload. This requires an even distribution of tuples to fragments and an even redistribution of resulting tuples into fragments at each iteration. Such situation can be produced only with a uniform distribution of values within join columns, while skewed distributions are likely to produce unbalanced fragments. A problem which is common to both approaches is that of duplicate elimination. The evaluation method described in [31] lacks a global union of deltas D_i, this means that duplicate tuples are not detected. In [12], a local union is performed with all incoming tuples of D_i, thus detecting duplicates produced at the same iteration, but no global union is done. The presence of duplicates creates unnecessary redundant computation.

Both [31] and [12] present an analysis of performance, based on a cost model described in [31]. This model assumes an architecture without shared memory (also called message-passing architecture). The time to produce new tuples is considered to be constant, and details about join and union algorithms are therefore not given. Relations are assumed to be acyclic. Analytic performance analysis, both in [31] and [12], demonstrate a strong advantage of the proposed methods in terms

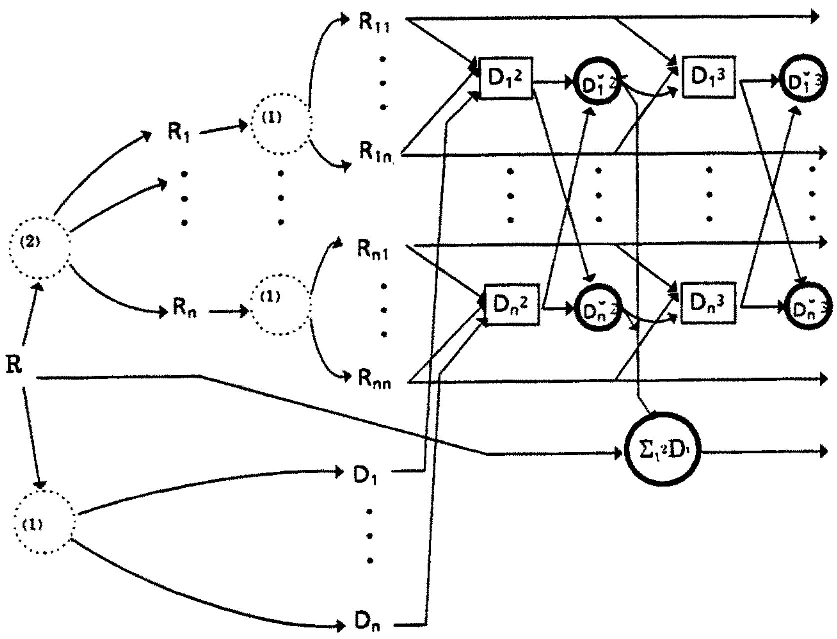

Figure 10: The method by de Maindreville and Cheiney

of computation speedup, thus confirming the intuition that hash-based fragmentation may be very efficient. However, these results rely heavily on the assumptions of uniform distribution and absence of duplicates within produced fragments; these assumptions do not hold in many applications. In the next section, we discuss an approach that uses a fragmentation derived from the semantics of the application domain.

3.4 Semantic fragmentation

The approach of Houtsma, Apers and Ceri, discussed in [17, 18, 20], builds a fragmentation specifically tailored to parallel execution of recursive queries. Such fragmentation, however, is also used in practice: the approach was suggested by real-world observations. Also in this case, the approach was developed in two stages; we describe the *disconnection set approach*, and then its generalization to *hierarchical fragmentation*.

3.4.1 Disconnection set approach

The basic idea that underlies the disconnection set approach can best be illustrated by an example. Consider a railway network connecting cities in Europe, and a question about the shortest connection between Amsterdam and Milan [1]

This question can be split into several parts: find a path from Amsterdam to the eastern Dutch border, find a path from the Dutch border to the southern German border, find a path from the German border to the Italian border, and find a path from the Italian border to Milan. We assume that data are naturally fragmented by state (e.g., there is a Dutch, a German, and an Italian database

[1]In pure Datalog this request cannot be expressed; however, several extensions of Datalog exist which provide the required features for this recursive query, among them the language Logres [5].

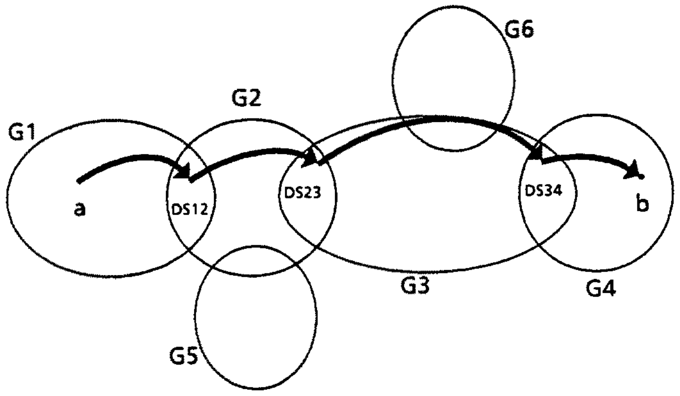

Figure 11: The disconnection set approach

which can be accessed through a common distributed database system). Moreover, we also assume that the points where one can cross a border are relatively few.

This fragmentation leads to a highly selective search process, consisting in determining the properties of connections from the origin to the first border, then between borders of two subsequent intermediate fragments, and finally from the last border to the destination city. These queries have the same structure; they apply only to a fragment of the database, and can be executed in parallel. The approach is sketched in Fig. 11.

The connection information is stored into a relation R; each tuple corresponds to an arc of the graph G, which can have cycles. By effect of the fragmentation, R is partitioned into n fragments $R_i, 1 \leq i \leq n$, each stored at a different computer or processor. This fragmentation induces a partitioning of G into n subgraphs $G_i, 1 \leq i \leq n$. *Disconnection sets* DS_{ij} are given by $G_i \cap G_j$. We assume that the number of nodes belonging to disconnection sets is much less than the total number of nodes in G.

In order to make the above approach feasible, it is required to store in addition some *complementary information* about the identity of border cities (i.e. the nodes in the disconnection set) and the properties of their connections; these properties depend on the particular recursive problem considered. For instance, for the shortest path problem it is required to precompute the shortest path among any two cities on the border between two fragments. This complementary information about the disconnection set DS_{ij} is stored at both sites storing the fragments i and j.

An important property of fragmentation is to be *loosely connected*: this corresponds to having an acyclic graph G' of components G_i. Formally, $G' =< N, E >$ has a node N_i for each fragment G_i and an edge $E_{ij} = (N_i, N_j)$ for each nonempty disconnection set DS_{ij}. Intuitively, if the fragmentation graph is loosely connected, then it is easier to select fragments involved in the computation of the shortest path between two nodes; in particular, for any two nodes, there is only one chain of fragments such that the first one includes the first node, the last one includes the last node, and remaining fragments in the chain connect the first fragment to the last fragment. However, for many practical problems (such as the European railway network) such a property does not hold.

In [18] it is shown that, if the fragmentation is loosely connected, then the shortest path connecting any two cities is found by involving in the computation only the computers along the chain of fragments connecting them [2]. Obviously, if the source and destination are within the same fragment, then the query can be solved by involving only the computer storing data about that fragment, in-

[2]Note that the shortest path might include nodes *outside* the chain, however their contribution is precomputed in the complementary information.

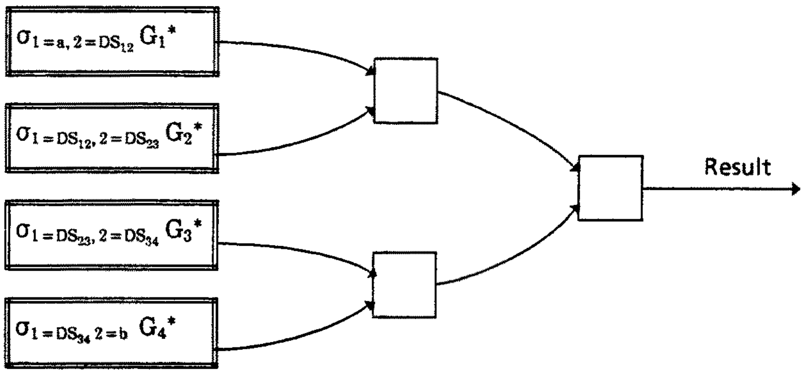
$\sigma_{1=a,2=DS_{12}} G_1^*$
$\sigma_{1=DS_{12},2=DS_{23}} G_2^*$
$\sigma_{1=DS_{23},2=DS_{34}} G_3^*$
$\sigma_{1=DS_{34}\,2=b} G_4^*$
Result

Figure 12: Example of query with the disconnection set approach

cluding all complementary information about disconnection sets stored at that fragment. In practice, this has the nice implication that queries about the shortest path of two cities in Holland can be answered by the Dutch railway computer system alone, even if the path goes outside the Dutch border. If instead the fragmentation is not loosely connected, then it is required to consider all possible chains of fragments independently for solving the query.

Along one chain of length n, query processing is performed in parallel at each computer; each subquery determines independently a shortest path (at the first fragment, between the start city and the first disconnection set; at the last fragment, between the last disconnection set and the destination city; at an intermediate fragment, between the input and output disconnection sets). Note that disconnection sets introduce additional selections in the processing of the recursive query, as they act as intermediate nodes that must be mandatorily traversed. These shortest paths are computed by considering also the complementary information. The final processing requires to combine all shortest paths obtained from the various processors with the complementary information, thus computing various "candidate" short paths, and selecting the shortest one among them. This process is shown in Fig. 12.

3.4.2 Parallel hierarchical evaluation

Houtsma, Cacace, and Ceri present in [21] a generalization of the disconnection set approach, called *parallel hierarchical evaluation*. This approach is inspired by real-life observations concerning transport problems. If we consider, for instance, the railroad network in Italy, we note that it is subdivided into geographical regions. Slow trains running inside regions stop at every station; inter-city trains connecting remote regions stop only at few stations, typically at the major cities of each region. Therefore, for long-distance travels, it makes sense to use the regional network around the departing city for connecting to the high-speed network of inter-city trains, then use inter-city trains, and finally use the regional network around the destination city in order to reach destination. This general rule may have an exception when start and destination cities are in adjacent fragments; in that case, the use of slow trains connecting the two regional networks may give the best solution.

The parallel hierarchical evaluation works exactly in this way. A subset of the connections are declared to be at high-speed, and form a fragment *HS*. All other connections are partitioned into n fragments $G_i, 1 \leq i \leq n$. The tuples of *HS* are stored on a separate computer together with complementary information about disconnection sets DS_i between *HS* and the various fragments G_i. Each fragment G_i is stored at a different computer together with complementary information of their disconnection sets. Any two fragments G_i and G_j are declared as either adjacent or nonadjacent. If two fragments G_i and G_j are *adjacent*, they are stored together with the complementary information

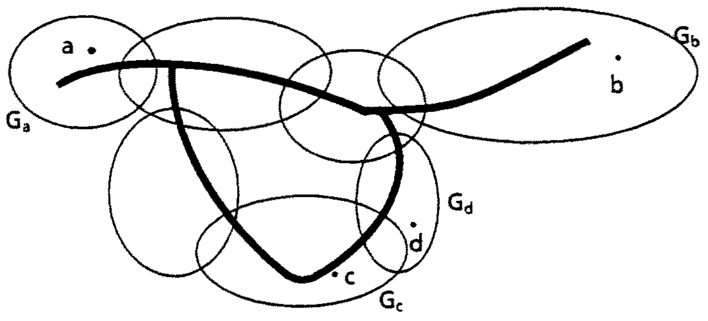

Figure 13: Parallel hierarchical evaluation

regarding their disconnection set DS_{ij}. Further, it is required that the shortest path between any two cities in G_i and G_j does not use connections of a different fragment, while it may use connections in *HS*. If two fragments G_i and G_j are *nonadjacent*, then the shortest path connecting any two cities of G_i and G_j should use the connections in G_i, G_j, and HS, but no other connection.

When these conditions hold, the computation of shortest paths connecting any two cities can be performed in parallel using three processors. If fragments are nonadjacent, the solution is found by composing shortest paths between the start city and DS_i in the processor storing G_i, between DS_i and DS_j in HS, and between DS_j and the destination city in G_j. If fragments are adjacent, then also the alternative of connecting the start city to DS_{ij} in G_i and DS_{ji} to the destination city in G_j must be evaluated and compared with the best solution which uses HS. Final post-processing for computing the total distance of the shortest path is performed exactly in the same way as in the disconnection set approach. An example of fragmentation with HS network is given in Fig. 13.

In [21], a procedure is defined for building a fragmentation that satisfies the applicability conditions of the method, starting from given centers of regions. Starting from these centers, an initial HS fragment is computed that includes the edges of the shortest paths between the centers. Then the fragments are gradually built by including edges with minimal distance from the centers into the appropriate fragments. Finally, the HS is suitably modified so that each pair of fragments is either adjacent or non-adjacent.

3.4.3 Concluding remarks

The disconnection set and the hierarchical approach are successful in partitioning the computation of one recursive query over a large relation R into several recursive queries over small relations R_i. One important speed-up factor is due to the reduced number of iterations required to compute each recursive query independently. Recall that the number of iterations required before reaching a fixpoint is given by the maximum diameter of the graph; if the graph is fragmented in n fragments G_i of equal size, then the diameter of each subgraph is highly reduced, hence giving efficient fixpoint evaluation.

Note that neither communication nor synchronization is required during the first phase of the computation; for evaluating the recursive subquery on a fragment any suitable single-processor algorithm may be chosen [1]; it is also possible to use some other parallel method. Only at the end of the computation, some communication is required for computing the final joins. These joins will have relatively small operands (since the disconnection sets are small) and pipelining may be used for their computation.

The disadvantage of the disconnection set approach is mainly due to the pre-processing required for building the complementary information and to the careful treatment of updates. Complementary information is different for each type of transitive closure query: in [19] the considered queries are connectivity, shortest path, and bill of material. However, as long as updates are not too complex and not too frequent, this cost may be amortized over many queries.

4 Conclusions

This paper has presented an overview of techniques for parallel evaluation of recursive queries. First we have shown the equivalence between transitive closure of relational expressions and a subclass of logic queries; this equivalence allows developing a general reference framework, where logic programming is used uniformly to express recursive programs and queries, and both *algebraic* and *logic* solution methods may be used to evaluate them. Furthermore, we have developed a graphical representation of algorithms, to show their characteristics when executed in parallel.

We have concentrated on giving an overview of algebraic methods for parallel execution of recursive queries (an overview of logic methods is included in [6]). Algebraic methods are essentially *evaluation* methods; they include the standard naive and semi-naive methods, but also several other methods. Some of them alter the basic structure of naive and semi-naive evaluation to build larger intermediate relations with fewer iterations; other methods use hash-based fragmentation to achieve parallelism; finally, other algebraic methods use semantic fragmentation to achieve efficient computation of special queries.

Processing of recursive queries is very hard; as such, recursive queries are particularly suited to parallel execution. The regular structure of recursion and the possibilities offered by hash-based or semantic fragmentation make parallel execution of recursive queries potentially very efficient.

References

[1] AGRAWAL R. AND JAGADISH H.V. "Direct algorithms for computing the transitive closure of database relations", in *Proc. 13th Int. Conference on Very Large Data Bases*, Brighton, 1987, pp. 255–266.

[2] APERS P.M.G., HEVNER A., AND YAO B., "Optimization algorithms for distributed queries", *IEEE-Transactions on Software Engineering*, SE9:1, 1983.

[3] APERS P.M.G., HOUTSMA M.A.W., AND BRANDSE F. "Processing recursive queries in relational algebra," in *Data and Knowledge (DS-2), Proc. of the 2nd IFIP 2.6 Working Conference on Database Semantics, Albufeira, Portugal, Nov. 3–7, 1986*, R.A. Meersman and A.C. Sernadas (eds.), North Holland, 1988, pp. 17–39.

[4] APERS P.M.G., KERSTEN M. L., AND H. OERLEMANS, "PRISMA database machine: a distributed main-memory approach," in *Advances in Database Technology*, Proc. Int. Conference Extending Database Technology (EDBT), 1988, pp. 590–593.

[5] CACACE F., CERI S., CRESPI-REGHIZZI C., TANCA L., AND ZICARI R. "Integrating object oriented data modeling with a rule-based programming language", in *Proc. ACM-Sigmod Conference*, Atlantic City, May 1990, pp. 225–236.

[6] CACACE, F., CERI, S., AND HOUTSMA, M.A.W. "A survey of parallel execution strategies for transitive closure and logic programs," Technical Report university of Twente–Politecnico Milano, Nov. 1990. Submitted for publication.

[7] CERI S., GOTTLOB G., AND LAVAZZA L. "Translation and optimization of logic queries: the algebraic approach", *in Proc. of the 12th Int. Conf. on Very Large Data Bases*, Kyoto, pp. 395-403, August 1986.

[8] CERI S., GOTTLOB G., AND TANCA L. *Logic Programming and Databases*, Springer-Verlag, 1990.

[9] CERI S., GOTTLOB G., TANCA L., AND WIEDERHOLD G., "Magic Semi-joins", *Information Processing Letters*, 33:2, 1989.

[10] CERI S. AND PELAGATTI G. *Distributed Databases: Principles and Systems*, Computer Science Series, McGraw-Hill, 1984.

[11] CERI S. AND TANCA L. "Optimization of systems of algebraic equations for evaluating Datalog queries," in *Proc. 13th Int. Conf. on Very Large Data Bases*, Brighton, Sept. 1987, pp. 31–42.

[12] CHEINEY J.P. AND DE MAINDREVILLE C. "A parallel strategy for transitive closure using double hash-based clustering," *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Aug. 1990, pp. 347–358.

[13] COPELAND G., ALEXANDER W., BOUGHTER E., AND KELLER T. "Data placement in Bubba," *Proc ACM-Sigmod Conference*, 1988, pp. 99-108.

[14] DE WITT D. J., GHANDEHARIZADEH S., AND SCHNEIDER D. "A performance analysis of the Gamma database machine," *Proc. ACM-Sigmod Conference*, 1988, pp. 350–360.

[15] GANGULY S., SILBERSCHATZ A., AND TSUR S. "A framework for the parallel processing of Datalog queries," *Proc. ACM-Sigmod Conference*, Atlantic City, May 1990, pp. 143–152.

[16] GOODMAN N., BERNSTEIN P.A., WONG E., REEVE C.L., ROTHNIE J.B. "Query processing in SDD-1 - A system for Distributed Databases", *ACM-Transactions on Database Systems*, 6:4, 1981.

[17] HOUTSMA M.A.W. *Data and Knowledge Base Management Systems: Data Model and Query Processing*, Ph.D. Thesis, University of Twente, Enschede, the Netherlands, Nov. 1989.

[18] HOUTSMA M.A.W., APERS P.M.G., AND CERI S. "Distributed transitive closure computation: the disconnection set approach," *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Aug. 1990, pp. 335–346.

[19] HOUTSMA M.A.W., APERS, P.M.G., AND CERI S. "Parallel computation of transitive closure queries on fragmented databases," Technical report INF88-56, University of Twente, the Netherlands, Dec. 1988.

[20] HOUTSMA, M.A.W., APERS, P.M.G., AND CERI, S. "Complex transitive closure queries on a fragmented graph," *Proc. 3rd Int. Conf. on Database Theory*, Lecture Notes in Computer Science, Springer-Verlag, Dec. 1990.

[21] HOUTSMA M.A.W., CACACE F., AND CERI S. "Parallel hierarchical evaluation of transitive closure queries," in preparation.

[22] HULIN G., "Parallel processing of recursive queries in distributed architectures," *Proc. 15th Int. Conf. Very Large Data Bases* , Amsterdam 1989, pp. 87–96.

[23] IOANIDIS Y. "On the computation of the transitive closure of relational operators," *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto 1986, pp. 403–411.

[24] KERSTEN, M.L., APERS, P.M.G., HOUTSMA, M.A.W., VAN KUIJK, H.J.A., AND VAN DE WEG, R.L.W. "A distributed, main-memory database machine," in *Proc. of the 5th Int. Workshop on Database Machines*, Karuizawa, Japan, Oct. 5–8, 1987; and in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka (eds.), Kluwer Academic Publishers, 1988, pp. 353–369.

[25] KLEINHUIS G. AND OSKAM K.R. "Evaluation and simulation of parallel algorithms for the transitive closure operation," M.Sc. Thesis, University of Twente, the Netherlands, May 1989.

[26] NEJDL W., CERI S., AND WIEDERHOLD G. "Evaluating recursive queries in distributed databases," Tech. Rep. 90-015, Politecnico di Milano, submitted for publication.

[27] RASCHID L. AND SU S.Y.W. "A parallel strategy for evaluating recursive queries," *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto 1986, pp. 412–419.

[28] SCHNEIDER D. A. AND DE WITT D. J. "A performance analysis of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proc. ACM-Sigmod Conference*, 1989, pp. 110–121.

[29] TANDEM DATABASE GROUP, "NonStop SQL: a distributed, high-performance, highly-availability implementation of SQL", in *High Performance Transaction Systems*, Lecture Notes in Computer Science, Springer-Verlag, 1987.

[30] ULLMAN J.D. *Principles of Data and Knowledge-Based Systems*", Computer Science Press, 1989.

[31] VALDURIEZ P. AND KHOSKAFIAN S. "Parallel Evaluation of the Transitive Closure of a Database Relation," Int. Journal of Parallel Programming, 17:1, Feb. 1988.

[32] VAN GELDER A., "A message passing framework for logical query evaluation," in *Proc. ACM-Sigmod Conference*, 1986, pp. 155–165.

[33] WOLFSON O. "Sharing the load of logic program evaluation," *Int. Symp. on Database in Parallel and Distributed Systems*, Dec. 1988, pp. 46–55.