Distributed and Parallel Databases 1 (1993), 337–382

A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs

FILIPPO CACACE, STEFANO CERI CERI@IPMEL2.ELET.POLIMI.IT
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy

MAURICE HOUTSMA HOUTSMA@CS.UTWENTE.NL
Department of Computer Science, University of Twente, 7500 AE Enschede, The Netherlands

Received April 30, 1993; Revised May 6, 1993

Recommended by: Patrick Valduriez

Abstract. An important feature of database technology of the nineties is the use of parallelism for speeding up the execution of complex queries. This technology is being 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 order to perform selections and joins in parallel. With the development of new query languages, and in particular with the definition of transitive closure queries and of more general logic programming queries, the new dimension of recursion has been added to query processing. Recursive queries are complex; at the same time, their regular structure is particularly suited for parallel execution, and parallelism may give a high efficiency gain. We survey the approaches to parallel execution of recursive queries that have been presented in the recent literature. We observe that research on parallel execution of recursive queries is separated into two distinct subareas, one focused on the transitive closure of Relational Algebra expressions, the other one focused on optimization of more general Datalog queries. Though the subareas seem radically different because of the approach and formalism used, they have many common features. This is not surprising, because most typical Datalog queries can be solved by means of the transitive closure of simple algebraic expressions. We first analyze the relationship between the transitive closure of expressions in Relational Algebra and Datalog programs. We then review sequential methods for evaluating transitive closure, distinguishing iterative and direct methods. We address the parallelization of these methods, by discussing various forms of parallelization. Data fragmentation plays an important role in obtaining parallel execution; we describe hash-based and semantic fragmentation. Finally, we consider Datalog queries, and present general methods for parallel rule execution; we recognize the similarities between these methods and the methods reviewed previously, when the former are applied to linear Datalog queries. We also provide a quantitative analysis that shows the impact of the initial data distribution on the performance of methods.

Keywords: Recursion, parallel algorithms, query optimization, deductive databases

1. Introduction

During the past decade, the execution of queries in relational databases has been improved through the use of multiple techniques, such as the use of

efficient physical data structures, the clear separation between *clients* (embedding all application-specific software) and *servers* (responsible for efficient database access), and the use of multitasking and multithreading within advanced architectures for database servers. All these techniques can be supported on a conventional, single processor architecture, but they are maximally exploited by multiprocessor architectures. These are becoming widespread, both in the context of specialized multiprocessor machines and in the context of distributed systems using either local or geographical computer networks. Indeed, database access is particularly suited for distributed, parallel execution, because it takes advantage of both *data* and *processing distribution*.

Parallelism in databases can be viewed from two different perspectives. *Interquery 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, 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. We will concentrate instead on the second type of parallelism, called *intraquery parallelism*, which enables the distribution of complex queries to multiple processors. Intraquery parallelism aims at minimization of the response time required for answering the query and at sharing, on multiple processors, of the heavy processing load required for its execution. In this case, each processor is typically dedicated to the query.

Intraquery parallelism has always been considered as an important feature in relational query optimization. Optimizers 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, intraquery parallelism has been considered as an underlying, implicit assumption of all the 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 [6, 25].

Data fragmentation is an essential ingredient for parallelism, as it enables a very natural partitioning of query processing. Each relation is partitioned into fragments that are stored on different disks, under the control of different processors. With this architecture, it is possible to assign the execution of relational operations, such as selections, projections, and joins, to several processors working in parallel [15]. Several experimental parallel systems based on data fragmentation were recently developed, including the *Prisma* machine, developed at Philips [8], the *Gamma* machine, developed at Wisconsin University [22, 54], the *Bubba* machine, developed at MCC [18], and the SIMD Relational Algebraic Processor, developed at the University of Essex [50]; only few commercial systems support fragmentation for intraquery parallelism, including Teradata [57] and Tandem [56].

In recent years, recursive queries have emerged as a new class of complex queries. These queries enable solving classical database problems, such as the *bill-of-material* (finding the number of elementary components of a given part). In all these problems it is assumed that a large *base relation* stores information about a binary relationship and enables building its transitive closure, possibly annotated with aggregate functions. In commercial applications, these problems are typically managed by embedding queries within programs; however, such applications are both hard to program and inefficient.

Database languages of the future will be able to express simple types of recursion within their query languages; 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 relational operations until termination (fixpoint) conditions are met. Though in most applications the termination of computation is certain, the number of required iterations may be very large and is not known a priori. Thus, intraquery parallelism is particularly needed for recursive queries.

1.1. Outline

In this paper, we survey parallel techniques for executing recursive queries. Section 2 presents some preliminary material, and, in particular, introduces the notion of transitive closure for algebraic expressions and shows how this notion can be useful for solving Datalog queries. Section 3 reviews the methods for sequential evaluation of transitive closure, distinguishing between iterative and direct methods; the former apply to a tabular representation of the base relation, the latter apply to a matrix-based representation.

The following sections address parallel execution methods for transitive closure. Section 4 presents a classification of the methods. Parallelism may be achieved by assigning operations to processors or by data fragmentation; fragmentation itself may be based on the use of hash functions or semantic criteria. These possibilities give rise to several classes of solution methods; each class is later separately analyzed.

Section 5 introduces a graphic formalism for describing parallel executions that combines relational algebra and some control operators. Section 6 introduces parallelism for iterative methods by assigning each algebraic operation to a distinct processor. Data fragmentation is introduced next; Section 7 deals with hash-based data fragmentation for iterative methods, while Section 8 deals with row-based fragmentation for direct methods. Section 9 deals with semantic data fragmentation.

Finally, in Section 10 we describe parallelization methods for general Datalog queries, distinguishing between program-oriented and rule-oriented methods. Program-oriented methods generate several Datalog programs and assign each of them to a different processor; rule-oriented methods assign each rule to

a different processor. We describe similarities between these methods, when applied to linear rules, and the methods for evaluating transitive closure.

An evaluation of the performance of the above transitive closure methods is beyond the scope of this survey, as it requires the comparison of techniques that are structurally very different and whose efficiency is highly influenced by data value distribution within relations and by the system architecture (e.g., computer's performance, network topology, cost of resources). Thus, we have restricted our performance analysis to one method (selected in the context of hash-based fragmentation), and have performed an in-depth analysis of the impact of initial data distribution of the proposed methods; this analysis is presented in Section 11.

2. Transitive closure and recursive queries

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. This is not surprising, since the semantics of Datalog programs (that is, pure Horn clauses without function symbols) can be expressed as Relational Algebra equations under a fixpoint semantics [33, 37]. This ensures the applicability of bottom-up evaluation and optimization methods to both recursive algebraic expressions and logic rules. Examples of research on algebraic optimization may be found in [5, 7, 10, 13, 16, 26, 37]; examples of logic optimization may be found in [11, 12, 39, 41, 42, 43, 47, 48, 51, 52, 68].

In this section we recall results about the use of the transitive closure operator for answering recursive queries. Though the material of this Section is self-contained, a general knowledge of the Datalog language might be useful, as it can be achieved through the reading of [14, 44, 58].

Consider a function-free Horn clause of the form

$$P_0(\vec{x}^{(0)}) \leftarrow P_1(\vec{x}^{(1)}) \wedge P_2(\vec{x}^{(2)}) \wedge \cdots \wedge P_k(\vec{x}^{(k)})$$

where, for every i, $\vec{x}^{(i)}$ is a subset of some fixed set of variables $(x_1, \ldots, x_n)$. We say that formula is *recursive* if, for some i, $P_i = P_0$. We are particularly interested in *linear* recursive clauses, where the predicate P_0 occurs exactly once in the right side of the clause. We further concentrate our attention on a program consisting of two clauses, one nonrecursive and one linear recursive. The program is therefore

$$\begin{aligned} &P(\vec{x}^{(r)}) \leftarrow Q_0(\vec{x}^{(0)}) \\ &P(\vec{x}^{(r)}) \leftarrow P(\vec{x}^{(n)}) \wedge Q_1(\vec{x}^{(1)}) \wedge \cdots \wedge Q_k(\vec{x}^{(k)}) \end{aligned} \tag{1}$$

The algebraic interpretation of this pair of logic clauses can be stated as follows [37]. We consider the relations corresponding to the *extensions* of predicates Q_i. The bottom-up computation of (1) yields at the first iteration $P^1 = Q_0$,

at the second iteration $P^2 = Q_0 \cup \pi_{\vec{x}(r)}(Q_0 \bowtie Q_1 \bowtie \cdots \bowtie Q_k)$, and so on. Join conditions, here omitted, force the equality of columns in correspondence to the same variables x_i in (1); thus, joins in relational algebra correspond to unification of variables appearing in the clauses. If we denote by $A_{Q_i}(X)$ the expression $(X \bowtie Q_1 \bowtie \cdots \bowtie Q_k)$, where X is a relation of suitable arity, then the relation P corresponding to the extension of the predicate P in (1) can be computed as

$$P = Q_0 \cup A_{Q_i}(Q_0) \cup A_{Q_i}(A_{Q_i}(Q_0)) \cup \cdots$$

If we further denote as jth power of A_{Q_i} the application of A_{Q_i} j times, we have

$$P = \bigcup_{j=0}^{\infty} A_{Q_i}^j(Q_0) \tag{2}$$

The above operation, transforming Q_0 into P, is called the *transitive closure* of A_{Q_i}.

In [37] an algebraic framework for the study of recursion is developed, showing how general recursion can be expressed as the fixpoint of an appropriate system of relational equations. The expressive power of first-order logic with transitive closure with respect to various Datalog extensions is investigated in [19].

The simplest and by far the most used Datalog program with the structure displayed in (1) is

$$\begin{aligned} &P(X, Y) :- R(X, Y) \\ &P(X, Y) :- R(X, Z), P(Z, Y) \end{aligned} \tag{3}$$

In this case, let $A_R(X) = \pi_{1,4}(R \bowtie_{2=1} X)$; the result relation P can be computed as:

$$P = R \cup \pi_{1,4}(R \underset{2=1}{\bowtie} R) \cup \pi_{1,4}(R \underset{2=1}{\bowtie} \pi_{1,4}(R \underset{2=1}{\bowtie} R)) \cup \cdots = \bigcup_{j=0}^{\infty} A_R^j(R)$$

If R denotes a parent-child relationship, then P computes the corresponding ancestor relationship. More in general, if the tuples of R denote a set of directed arcs $\langle n_i, n_j \rangle$ in a graph G, then P gives the pairs of nodes $\langle n_h, n_k \rangle$ such that there is a path in G from n_h to n_k.

The graph interpretation is useful to classify different kinds of instances for the base relation R. There are four cases of interest: the graph can be a *list* (nodes form a directed chain), a *tree* (each node has more than one immediate successor, but only one immediate predecessor), a *DAG* (directed acyclic graph) (nodes can have more than one immediate predecessor, but no cycles are present), or a *cyclic graph*.

Some of the algorithms for computing the transitive closure deal only with a specific type of graph. Moreover, the form of the graph has a relevant impact on the performance of algorithms, particularly in the parallel case. For

example, DAGs and cyclic graphs produce redundant paths that may cause loss of performance, especially with multiprocessor environments, as we shall see in the sequel.

The graph interpretation highlights the analogies of the computation of the transitive closure of a relation with the broader area of *path problems*. Actually, many database query problems involving the computation of transitive closure may be regarded as path problems [1, 46]. Examples of such problems are reachability, shortest path, maximum capacity path, and bill of materials. Algorithms for the computation of transitive closure can be extended to deal with this important class of problems. Termination of the computation of path problems with cyclic graphs and positive labels is guaranteed for *absorptive* problems, including reachability, maximum capacity path, and most reliable path [20].

The graph interpretation of the transitive closure suggests as well a *matrix representation* of the input data. The square matrix has as many rows and columns as the number of different values of the source and target fields in the original relation. Several algorithms, called *direct algorithms*, use stored data in this format rather than in a relational format.

The last important discriminant for transitive closure algorithms is the kind of queries they are able to answer. Queries on recursively defined relations can contain constants; in this case, it is important for the evaluation algorithm to be able to exploit their presence in order to reduce the amount of computation needed to answer the query [2, 34, 21].

The presence of constants in the query is described by an *adornment* of the corresponding predicate. An adornment is a mapping from the set of arguments of the predicate to the set $\{b, f\}$ (where b means "bound" and f means "free"); the adornment of a predicate argument is b if that argument is a constant value. For example, the query $? - P(a, Y)$ has the adornment P_{bf}. While a query with adornment P_{ff} requires the complete evaluation of the transitive closure P, queries with adornment P_{bf}, P_{fb}, and P_{bb} may not require a complete evaluation. The straightforward approach for solving these queries is to apply a final selection on the result. However, the most efficient method—only applicable for some transitive closure expressions—consists of anticipating the selection [5, 14, 39]. For instance, query $? - P(X, a)$ for program (3) can be solved as follows:

$$P_{fb} = \sigma_{2=a}\left(\bigcup_{j=0}^{\infty} A_R^j(R)\right) = \bigcup_{j=0}^{\infty} A_R^j(\sigma_{2=a}R)$$

This formula allows computing the first join by using $\sigma_{2=a}R$ as an operand, instead of R. Its applicability, proven in [5], stems from the fact that the original program (3) is right-recursive; hence a selection on the second column can be distributed to the right operand of a chain of joins. The query $? - P(a, Y)$ cannot be simplified in the same way; however, the same query can be applied to the following left-recursive program (4), equivalent to (3):

$$P(X, Y) :- R(X, Y)$$

$$P(X, Y) :- P(X, Z),\ R(Z, Y) \tag{4}$$

In this case, let $A_R(X) = \pi_{1,4}(X \bowtie_{2=1} R)$; then we have

$$P_{bf} = \sigma_{1=a}\left(\bigcup_{j=0}^{\infty} A_R^j(R)\right) = \bigcup_{j=0}^{\infty} A_R^j(\sigma_{1=a}R)$$

As in the previous case, the above formula allows distributing the join to the first operand. Thus, the first join can be evaluated by using $\sigma_{1=a}R$ as an operand, instead of R. More general conditions for the application of selections during the evaluation of transitive closure are given in [21].

3. Sequential methods for transitive closure

In this section we present an overview of the most relevant sequential algorithms proposed in literature for the computation of the transitive closure of a relation. The parallel methods reviewed in the Sections 6–9 use these methods as their basis.

Sequential methods fall in one of two categories: *iterative* and *direct* algorithms. The essential idea behind iterative algorithms is to evaluate the transitive closure breadth-first, with a loop containing algebraic expressions that derive new tuples, until no new element is generated.

Direct algorithms use the matrix representation of a graph introduced in the previous subsection; they operate depth-first. They require considering each node of the graph a fixed number of times, independently of the structure of the graph; in contrast with iterative methods where the number of iterations is a priori unknown. These algorithms are called "direct" in the sense that they exploit the special structure of the transitive closure problem rather than solving general recursion. Some direct algorithms may solve the reachability problem, but do not solve more complex path problems (i.e., shortest path, bill-of-materials, etc.) [53]. We shall only consider direct algorithms that can solve path problems.

3.1. Iterative algorithms

Iterative algorithms include naive, semi-naive, squaring, smart, and minimal evaluations.

3.1.1. Naive evaluation. Naive evaluation is the simplest bottom-up, breadth-first strategy [10]; it directly applies formula (2) to compute the transitive closure. The algorithm is shown in Figure 1. At each iteration i, the $(i+1)$st power of R is computed giving all the paths of length up to $(i+1)$ in the graph, by joining the ith power of R with R, and then performing the union of all previous results with the $(i+1)$st power of R. This process is iterated until the ith iteration does

```
power := R;
union := R;
repeat
        old_union := union;
        power := π (power ⋈ R);
        union := union ∪ power
until (union − old_union) = ∅
```

Figure 1. Naive algorithm.

```
delta := R;
union := R;
repeat
        power := π(delta ⋈ R);
        delta := power − union;
        union := union ∪ delta
until  delta = ∅
```

Figure 2. Semi-naive algorithm.

not add new tuples to the union of previous results. Three algebraic operations are involved: joins compute subsequent powers of R, unions gather them, and set difference tests for the termination of the computation. Note that for cyclic graphs and DAGs, the same tuple may be produced in several iterations, thus yielding redundancy.

3.1.2. Semi-naive evaluation. A simple variation of naive evaluation is semi-naive evaluation[1] [9–11]. The idea is to use, at each iteration, only the new tuples derived at the previous iteration (denoted as *delta*) in order to compute the subsequent *power*. This reduces the amount of redundant computation introduced by naive evaluation. The *delta* can be determined easily by computing the difference between tuples computed at the ith iteration and tuples computed at previous iterations, as shown in Figure 2.

With respect to naive evaluation, semi-naive introduces significant potential for increased efficiency, because the cardinalities of relations involved in the joins are reduced.

3.1.3. Squaring evaluation. Squaring evaluation, introduced in Apers et al. [7], is based on the idea of reducing the number of iterations rather than the cardinality of the operators. In this method, the result from the previous iteration is squared at each step. Hence, first paths of length up to 2 are computed, then paths of lengths up to 4, then paths of length up to 8, and so on. As in the case

```
union := R;
repeat
        old_union := union;
        union := union ⋈ union;
        union := union ∪ R
until (union − old_union) = ∅
```

Figure 3. Squaring algorithm.

```
delta := R;
union := R;
power := R;
repeat
        delta := π (delta ⋈ delta);
        power := π (union ⋈ delta);
        union := union ∪ delta ∪ power
until power = ∅
```

Figure 4. Smart algorithm.

of naive evaluation there is a substantial amount of redundant computation for cyclic graphs. This method is shown in Figure 3.

3.1.4. Smart evaluation. "Smart" (or "logarithmic") evaluation, introduced by Ioannidis [35] and Valduriez [60], uses an improved variation of the squaring approach, by considering at each iteration the paths of length 1, 2, 4, 8, etc., to create paths of length 3, 5, 6, 7, etc. (see Figure 4). Also this method produces redundant computations.

3.1.5. Minimal evaluation. Several other rewritings of the transitive closure operation are introduced by Ioannidis [35]. One of them is the "minimal" algorithm, so called because it requires the minimal number of operations for computing the transitive closure. Obviously, these operations are generally very complex, so the reduction in number of operations is outbalanced by their increasing complexity; moreover, the redundant computation is still present. This method is shown in Figure 5.

3.2. Direct algorithms

Direct algorithms were initially proposed by Warshall and Warren; these were applied to main-memory representations of an adjacency matrix. Variants of

```
delta := R;
union := R ∪ (R ⋈ R);
power := R;
repeat
        delta := π (delta ⋈ delta ⋈ delta);
        power := π ((union ⋈ delta) ∪ (union ⋈ (delta ⋈ delta)));
        union := union ∪ delta ∪ (delta ⋈ delta) ∪ power
until power = ∅
```

Figure 5. Minimal algorithm.

```
Input: a v × v Boolean matrix of elements a_ij, with a_ij being 1 if
there is an arc from node i to node j and 0 otherwise
    For i=1 to v
        For k = 1 to i − 1
            For j = 1 to v
                a_ij = a_ij ∨ (a_ik ∧ a_kj);
    For i = 1 to v
        For k = i + 1 to v
            For j = 1 to v
                a_ij = a_ij ∨ (a_ik ∧ a_kj);
```

Figure 6. Warren's algorithm.

these methods [1, 36] use the notions of mass-memory blocks to improve the performance of computations when data are stored in mass memory instead.

3.2.1. Warshall and Warren algorithms. Algorithms to compute the transitive closure of the adjacency matrix of a graph have been initially proposed by Warshall [65] and Warren [64]. In their original formulation they are tuple-oriented and depth-first. Since Warren's algorithm is generally more efficient than Warshall's, we illustrate the former algorithm only. The computation can be expressed as in Figure 6 (this formulation is due to [1]).

The initial relation is represented as an *adjacency matrix*: if the tuple (i, j) belongs to the relation, the value of a_{ij} is 1, otherwise it is 0. For each node i, its successor list is fetched (the row a_{ij} in the matrix), and for each successor k such that $a_{ik} = 1$ its successor list (row a_{kj}) is fetched and added to the successor list of i. The algorithm requires two "passes" on the matrix to complete. Successors $k < i$ are examined in the first pass and successors $k \geq i$ in the second pass. The process terminates after a fixed number of iterations, depending on the dimension of the square matrix.

Note that the adjacency matrix is a square matrix, having as many rows and columns as the different values of the source and target fields in the relation on

```
/* First Pass */
For each row partition (rows i_b to i_e inclusive)
    For j = 1 to i_e
        For i = i_b to i_e
            If tuple ⟨i, j⟩ exists
                Add the successor list of j to successor list of i
/* Second Pass */
For each row partition (rows i_b to i_e inclusive)
    For j = i_b to n
        For i = i_b to i_e
            If tuple ⟨i, j⟩ exists
                Add the successor list of j to successor list of i
```

Figure 7. Blocked Warren algorithm.

which the transitive closure is performed. For general path problems, the tuple (i, j, L_{ij}) (where L_{ij} is the label associated with the arc (i, j)) corresponds to a value L_{ij} for the element a_{ij}. A straightforward database implementation of the adjacency matrix for the Warren algorithm would require sorting the relation on the source attribute in order to build the successor list of each node i. All the "successors" of a given node can be found in a contiguous set of tuples in the sorted relation.

3.2.2. Blocked Warren algorithm. Agrawal and Jagadish consider the database implementation of the Warshall-Warren algorithms in [3]. Their objective is to provide a good implementation of these algorithms under the constraint that the entire relation cannot reside in main memory all at once. The matrix is therefore divided into *blocks* of rows, where each block is transferred from mass memory to main memory through an input-output operation. The algorithm from [3] is shown in Figure 7. The blocks are indicated through the pair (i_b, i_e), indicating the first and last row. The rationale behind the algorithm is to find an order of computation that satisfies the precedence constraints of Warren's algorithm and minimizes the amount of input-output (I/O) due to fetching successor lists (the rows of the matrix) not contained in the local partition.

In the first pass, a partition of rows (i_b, i_e) is examined at a time. In each partition only elements with column number $j \leq i_e$ are examined. The first pass proceeds columnwise within each partition: the order of processing is therefore $a_{ib,1}, a_{i_b+1,1}, \ldots, a_{i_e,1}, a_{i_b,2}, a_{i_b+1,2}, \ldots, a_{i_e,2}, \ldots, a_{i_b,i_e}, a_{i_b+1,i_e}, \ldots, a_{i_e,i_e}$. For each element a_{ij} being processed, if the element value is 1 it is necessary to fetch the list of successors of j. But this successor list corresponds to the jth row $a_{j,1}, \cdots, a_{j,v}$. Since all the elements on the same column have the same j, they need the same list of successors, and this reduces the amount of I/O for fetching successor lists. Moreover, each row in the partition will require the same set of successor lists, thus they can be read only once for each partition. Finally,

$(i_e - i_b)$ of the successor lists are already contained in the row partition and do not require extra I/O.

In the second pass the rest of the elements are examined. The processing is still columnwise and proceeds as before; moreover, the row partition is not necessarily the same as in the first pass.

4. A classification of parallel methods for transitive closure

In this section, we discuss various alternatives for introducing parallelism in the computation of the transitive closure.

The methods developed in the literature so far optimize queries "in isolation", rather than considering mixes of queries; thus, processors and channels among them are fully assigned to a single query. Parallelism can be achieved by

- Assigning different operations to different processors
- Assigning different data to different processors (data fragmentation)
- Combining the two cases above

Tuples produced by one processor are generally pipelined to the next processor along interprocessor channels. However, in some cases it is required to synchronize execution of processors more tightly; in this case tuples are stored in intermediate memories and processor execution is enabled by appropriate synchronization signals.

The option of assigning processors to specific operations applies to iterative methods, where set-oriented operations (such as join, union, test for equality) are clearly distinguished; this possibility is explored in Section 6.

Most methods published in the literature use data fragmentation. They can be further distinguished into

- Hash-based fragmentation, where tuples in a relation are partitioned into fragments on the basis of hash functions evaluated on one of the join columns or on both of them.
- Semantic fragmentation, where tuples are assigned to fragments on the basis of their semantic properties; for instance, topological properties: all tuples corresponding to edges of a particular subgraph are assigned to a given processor.

Hash-based fragmentation can be applied both to iterative and to direct methods. In the former case, joins performed at each iteration are also partitioned: each processor is assigned to the join portion corresponding to a specific fragment. In the latter case, the matrix is partitioned (e.g., rowwise) and each processor is assigned to a particular collection of rows. In both cases, each iteration produces

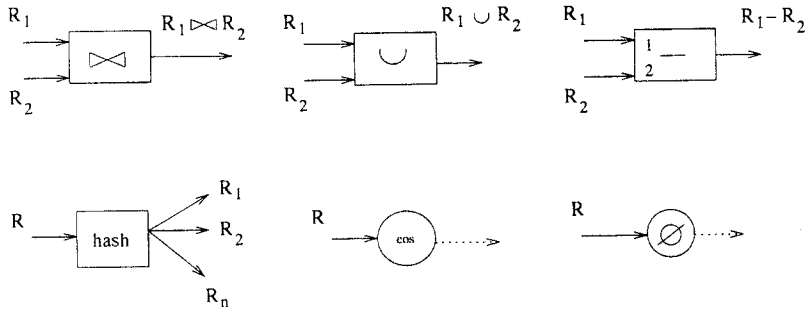

Figure 8. Elements of the graphical model.

results that need to be redistributed based on the hashing criterion. Iterative methods are discussed in Section 8, direct methods in Section 9.

Semantic fragmentation achieves parallelism by separating the tuples of relations into fragments so that each fragment can be independently considered; transitive closure is then computed by first computing the transitive closure within one fragment, then across fragments. By interpreting tuples as the edges of a graph, each fragment is mapped by semantic fragmentation to a subgraph that has maximal cohesion, while connections between subgraphs are loose. As such, semantic fragmentation can be considered as a metalevel method; it can be coupled with any other technique for the computation of transitive closure inside a fragment. Semantic fragmentation is discussed in Section 9.

5. Graphic representation of parallel algorithms for transitive closure

This section introduces a graphical model that helps in highlighting parallelism of computations. The model is applicable to iterative methods, which are set-oriented; it cannot describe direct methods, which are tuple-oriented.

Algorithms are represented through diagrams, similar to dataflow diagrams but with explicit synchronization operations. The elements of diagrams, shown in Figure 8, are

- Relational operators, represented by squares
- Synchronization operators, represented by circles
- Data flows, represented by solid arrows
- Synchronization messages, represented by dashed arrows

Relational operators are *join, union, difference*, and *hash*. Synchronization operators are the *empty* operator, which tests its input data flow in order to decide whether it is an empty relation, and the *eos* operator, which detects

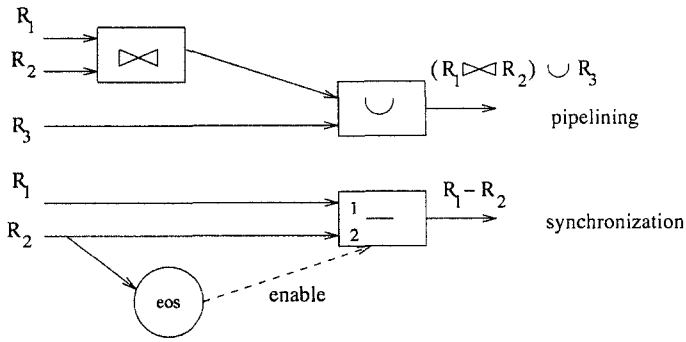

Figure 9. Examples of compositions of blocks.

the end of a stream of data tuples. Solid arrows indicate data flow among blocks; labels of arrows indicate relations or fragments being transmitted. For simplicity, we suppose that each data flow also carries an *end of flow* control message. Dashed arrows indicate synchronization signals, which are produced by synchronization blocks, and can be of two kinds: *stop* signals or *enable* signals (indicated by their labels). Synchronization signals are input to other blocks. The semantics of the stop signal is that of halting the process that executes the block, while the semantics of the enable signal is that of activating the process and start a block execution.

In Figure 9, two examples of block composition are shown. Pipelining is represented through chains of relational operators without synchronization signals (other than the implicit *end of flow* associated with data). Synchronous operations are enabled by appropriate signals. In Figure 9, the synchronization of a relational difference operation is illustrated: the operation is enabled as soon as the second operand is completely produced.

Note that with iterative algorithms the number of required iterations is not known a priori (though for the computation of *absorptive* problems it is known to be finite); we typically describe only two iterations. To simplify diagrams, we display termination control only for one iteration (typically, the second one).

6. Operations-to-processors mapping with iterative methods

Iterative algorithms described in Section 3.1 can be regarded as variations to naive evaluation that try to improve its performance by either reducing the number of tuples considered at each iteration (through the semi-naive approach) or by 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 of

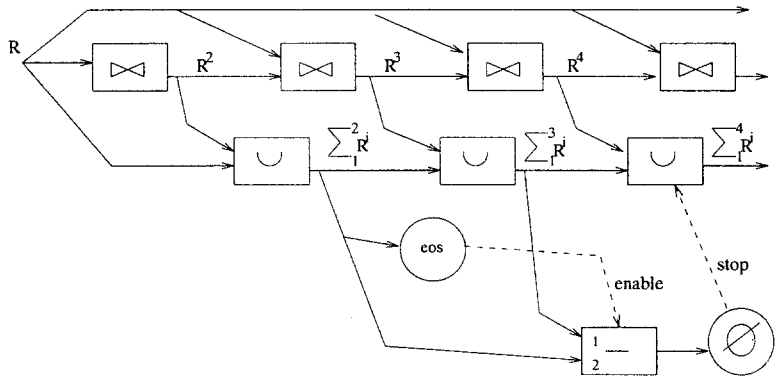

Figure 10. Naive evaluation.

this approach, thus enabling us to identify the features that lead to an inherently sequential behavior.

One way of introducing parallelism in the evaluation consists of assigning a specialized processor to each *type of operation*. For instance, one processor performs all joins, a second one performs all unions, a third one performs all differences. In this case, there is a strict serialization 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 to start each processor 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 testing termination, 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 in parallel with the evaluation of a termination test, a positive test outcome may become known after a large amount of additional processing has already occurred.

6.1. Naive evaluation

The graphic representation of the naive algorithm of Figure 1 is shown in Figure 10. 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. The figure highlights the regularity of the computation, which is iterated until a fixpoint is reached.

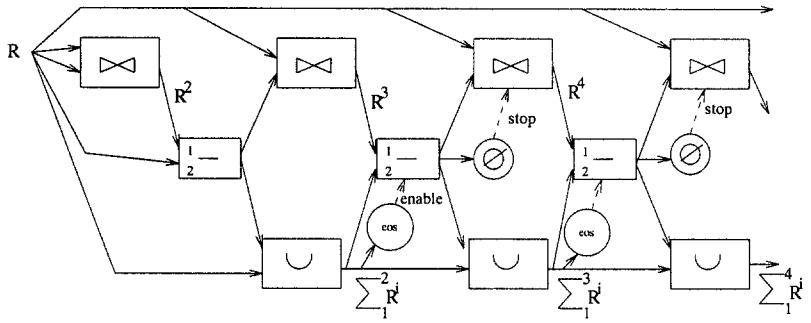

Figure 11. Semi-naive evaluation.

Unions, joins, and differences may each be assigned to a different processor or to a different class of processors.

Since set difference determines the termination of the computation, it acts as a synchronization point. Join processors at the ith iteration may be activated when the corresponding difference produces the first tuple (hence, when the difference fails as termination test); alternatively, joins can proceed asynchronously based on the pipelining of tuples, at the risk of performing unnecessary computations if the corresponding iteration is then halted by a successful termination test.

6.2. *Semi-naive evaluation*

Figure 11 shows the semi-naive evaluation of transitive closure queries, which was described in an algorithmic way, in Figure 1. At each iteration, set difference and join operations are interleaved, in order to eliminate duplicates. Therefore, join and set difference operations are strongly synchronized. The union that builds the final results may be done asynchronously on a separate processor, but this does not lead to major parallelism. In summary, this schema for semi-naive evaluation is intrinsically sequential.

We can make a general comment at this point. Semi-naive improves over naive evaluation by reducing the amount of data considered at each iteration. However, such reduction is performed by means of a set difference, which requires synchronization. Thus, there is a trade-off between maximizing the speedup of the computation and reducing the number of tuples processed by joins. Several parallelization methods, discussed in Section 7, disregard the use of set differences for reducing the number of tuples, and use them instead as termination tests, which are performed asynchronously.

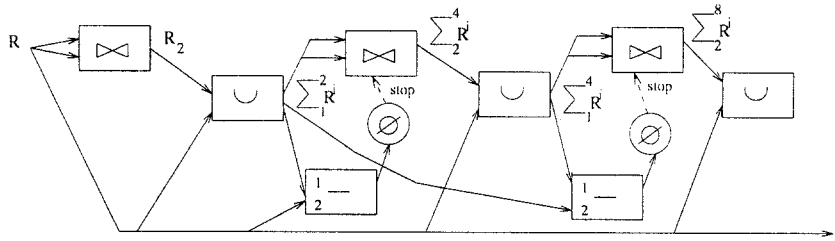

Figure 12. Squaring evaluation.

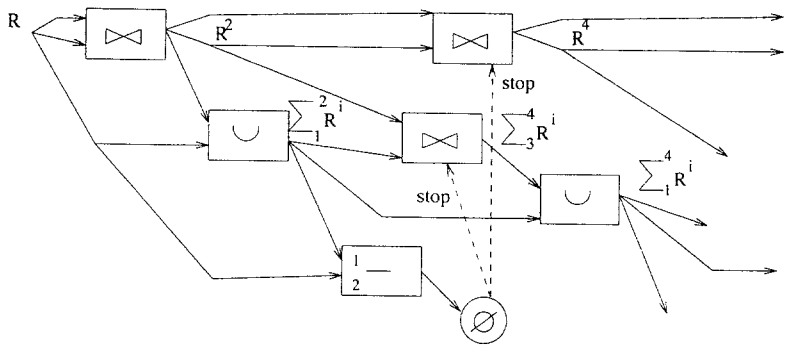

Figure 13. "Smart" evaluation.

6.3. Squaring, smart, and minimal evaluation

In the squaring, smart, and minimal evaluation, the number of iterations required to compute the transitive closure is reduced by immediate use of power relations during iterations. The graphical representation of the squaring evaluation, shown in Figure 12, shows that joins and unions are interleaved, while set differences are only performed for testing termination and can be done asynchronously. This schema is also used by smart and minimal evaluation, and is very similar to naive evaluation. In the smart and minimal evaluation, represented in Figures 13 and 14, unions must by synchronized with power joins.[2] The most serious drawback for algorithms square, smart, and minimal, is the inefficient handling of duplicate tuples that are produced with cyclic graphs.

6.4. Iterative algorithms for queries with one bound argument

The above algorithms were shown in the computation of the entire transitive closure. They can also be used for queries having one of the arguments of P bound to a constant (e.g., ?– $P(a, X)$ or ? – $P(X, a)$); let P_{bf} and P_{fb} denote the results of these queries.

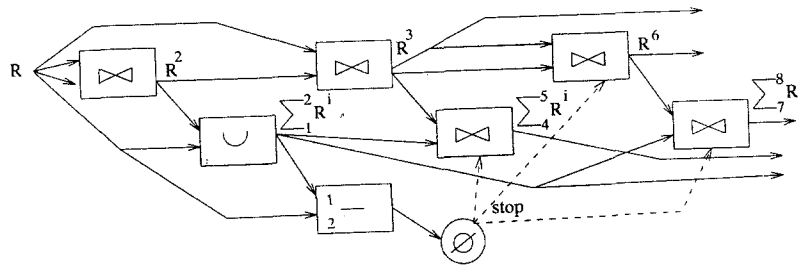

Figure 14. "Minimal" evaluation.

Naive and semi-naive evaluations allow to anticipate the selections, as shown in Section 2; instead, squaring, smart and minimal evaluation require the computation of complete powers of R (for example, R^2 is needed for evaluating R^4 as $R^2 \bowtie R^2$); the anticipation of the selection is not straightforward.

6.5. *An iterative algorithm for queries with two bound arguments*

In this section we discuss a parallel version of naive evaluation as proposed by Raschid and Su [49] for the case of queries with two bound arguments. They consider again the recursive program (3), but they concentrate on the query $? - P(a, b)$, characterized by having both arguments bound to a constant; let P_{bb} denote the relation produced as result. Note that, for a particular choice a and b of bindings, P_{bb} returns the tuple (a, b) if $P(a, b)$ can be proved, the empty relation otherwise. We denote with P^i_{bb} the tuples produced at the ith iteration for solving the P_{bb} query, similarly for P^i_{bf} and P^i_{fb}; finally, the summation $\sum_1^k P^i_{bb}$ gives all the result tuples for the query P_{bb} after k iterations.

Figures 15 and 16 show the algorithm of Raschid and Su and its graphical representation. Though in principle this architecture is capable of solving the P_{bf} and P_{fb} queries, in practice the architecture degenerates to the naive evaluation in those cases[3]. Thus, this architecture applies successfully only to the P_{bb} query.

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

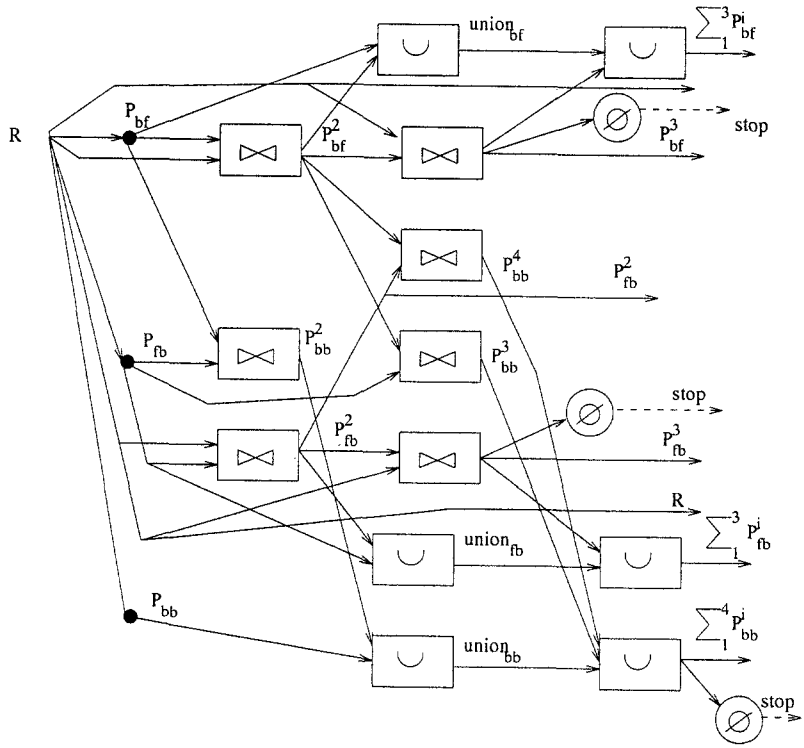

Figure 15. The method by Raschid and Su.

At each iteration, three union operations are also required: $\bigcup_i P^i_{bf}$, $\bigcup_i P^i_{fb}$, and $\bigcup_i P^i_{bb}$. The query P_{bb} is solved when $\bigcup_i P^i_{bb}$ produces one tuple. It is also solved negatively, in the sense that the answer is the empty relation, when both P_{fb} and P_{bf} are 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. Therefore, the suggested optimal number of processors is seven.

In this approach, parallelism comes together with massive interprocessor communication, and this is rather costly. In [40] it was reported that simulations of this method on a model of the *Prisma* database machine actually showed a negative speedup, due to synchronization and communication costs. This parallel strategy turned out to be slower than smart single-processor strategies.

$P^1_{bf} := \sigma_{(1=a)}R$;
$P^1_{fb} := \sigma_{(2=b)}R$;
$P^1_{bb} := \sigma_{(1=a,2=b)}R$;
$\text{union}_{bf} := P^1_{bf}$;
$\text{union}_{fb} := P^1_{fb}$;
$\text{union}_{bb} := P^1_{bb}$;
$i := 0$;
if $P^1_{bb} \neq \emptyset$ **then**
  repeat
    $i := i + 1$;
    $P^{2i}_{bb} := P^i_{bf} \bowtie P^i_{fb}$;
    if $i > 1$ **then** $P^{2i-1}_{bb} = P^i_{bf} \bowtie P^{i-1}_{fb}$;
    $P^{i+1}_{bf} := P^i_{bf} \bowtie R$;
    $P^{i+1}_{fb} := R \bowtie P^i_{fb}$;
    $\text{union}_{fb} := \text{union}_{fb} \cup P^{i+1}_{fb}$;
    $\text{union}_{bf} := \text{union}_{bf} \cup P^{i+1}_{bf}$;
    $\text{union}_{bb} := \text{union}_{bb} \cup P^{2i}_{bb} \cup P^{2i-1}_{bb}$;
  until $\text{union}_{bb} \neq \emptyset$ **or** $((P^{i+1}_{fb} = \emptyset)$ **and** $(P^{i+1}_{bf} = \emptyset))$

Figure 16. Algebraic representation of the Raschid and Su algorithm.

7. Hash-based fragmentation

In this section we study methods for parallel execution of recursive queries that use hash-based fragmentation and semi-naive computation. The basic idea behind these strategies is to use fragmentation of relations R and S in order to compute $R \bowtie S$ as $R_1 \bowtie S_1 \cup \cdots \cup R_n \bowtie S_n$. This approach is called *simple distributed join* in [15], where applicability and correctness conditions are discussed.

We consider a fragmentation of R into $R_1, \cdots, R_n$ 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 Khoshafian [61], then we discuss extensions of this approach described by Cheiney and de Maindreville [17] and Agrawal and Jagadish [4].

7.1. *The method by Valduriez and Khoshafian*

The method by Valduriez and Khoshafian [61, 62] is shown in Figure 17.[4] 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

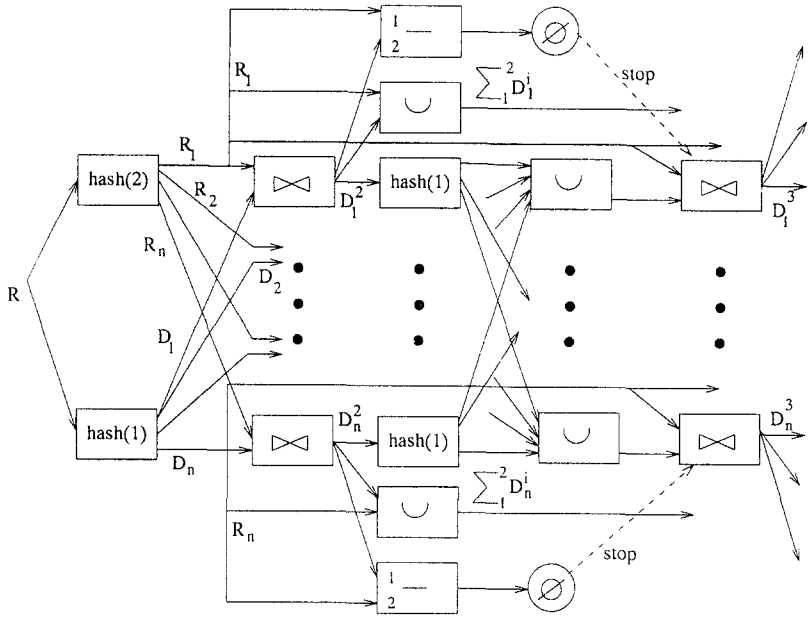

Figure 17. The method by Valduriez and Khoshafian.

and also distributed to these processors; this copy represents the *delta* relation.

This fragmentation has to be fully understood: 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 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: $D_i^2 := R \bowtie_{2=1} D_i$. 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 ($D_i^3 := R_i \bowtie_{2=1} D_i^2$). 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 (in different iteration steps), thus leading to unnecessary, redundant computations.

When R corresponds to a directed acyclic graph, the computation ends when all deltas are empty. When instead R corresponds to a relation with cycles, the union of the deltas produced in each iteration at all processors, and a set difference with the union produced at the previous iteration, are required for detecting termination. In Figure 17 we only show the accumulation of deltas at each processor; eventually, all accumulated deltas have to be gathered by a union operation (not shown in Figure 17).

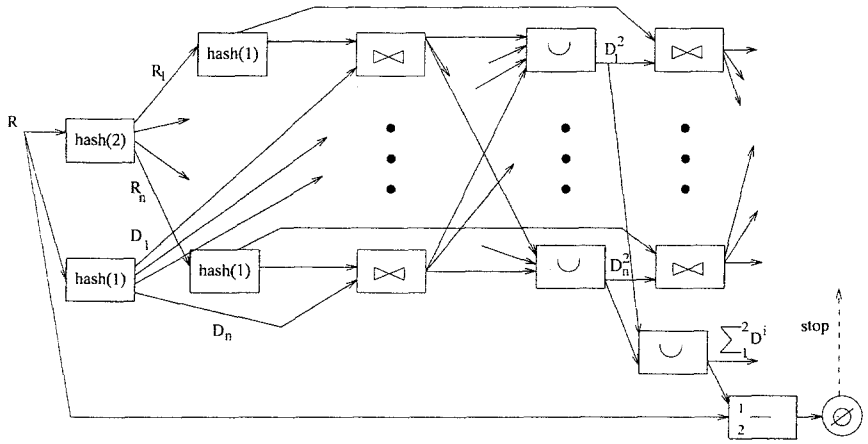

Figure 18. The method by Cheiney and de Maindreville.

7.2. *The method by Cheiney and de Maindreville*

In [17], Cheiney and de Maindreville show a simple extension of the evaluation strategy by Valduriez and Khoshafian that avoids rehashing deltas at each iteration step. This strategy is shown in Figure 18[5]. It differs from the approach described in [61] in one feature only: after hashing R on the second attribute, the resulting n fragments R_i are further hashed on their first attribute. Each R_i is thus conceptually divided into n-subfragments R_{ij}. After joining with the delta fragments at each iteration k, the second hashing is used to predetermine where tuples of delta relations D_i^k need to be sent, without need for their rehashing. Figure 18 shows this algorithm.

Note that this approach can be even further extended by assigning a processor to each subfragment 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.

7.3. *The method by Agrawal and Jagadish*

A third alternative for semi-naive hash-based evaluation has been proposed by Agrawal and Jagadish in [4]. The central idea is to eliminate interprocessor communication at the end of each iteration. In order to do so, the entire relation R is used by each processor as an operand of the join, the other operand being the result of the previous iteration.

The hashing phase (on the first attribute) takes place only at the first iteration, as in the case of the previous algorithm. All subsequent computation can be

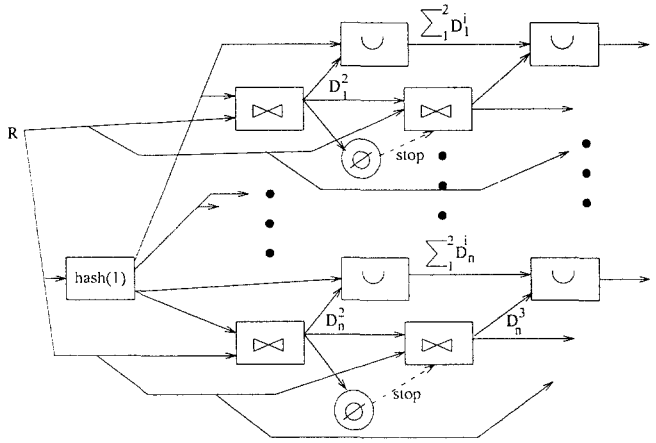

Figure 19. The method by Agrawal and Jagadish.

performed locally at each processor: no collection of tuples from the other processors is needed. In terms of the graph corresponding to the given relation, this algorithm assigns the complete graph to each processor, but makes a processor responsible for determining reachability from a specified set of nodes.

The algorithm is shown in Figure 19. Note that there is different halting condition for each process, which independently detects its own termination.

7.4. *Concluding remarks*

Approaches presented in this section extend to recursive query processing the work currently being done for applying *intraquery parallelism* to joins [18, 22, 56]. 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 a 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 that is common to the first two approaches is that of duplicate elimination. The evaluation method described in [61] lacks a global union of deltas D_i, this means that duplicate tuples are not detected. In [17], 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 approach of [4] is capable of distributing semi-naive evaluation and performing the termination test in parallel

on each processor; this factor may be significant, especially with DAGs or cyclic data, where many independent paths may cause redundant computations.

Both [61] and [17] present an analysis of performance, based on a cost model described in [61]. This model assumes an architecture without shared memory. 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 [61] and [17], demonstrates a strong advantage of the proposed methods in terms 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.

Experimental results were presented in [4] for the third algorithm of this section. The algorithm was tested both in a shared memory and in a shared-nothing architecture (where processors have a local memory); performances were very similar in the two cases. The relative speedup with respect to semi-naive execution, in the case of DAGs was almost linear with the number of processors and constant with respect to the size of the DAG. By using eight processors a speed-up of 6.9 was achieved, but performances for cyclic instances were not studied.

8. Direct algorithms

The parallel execution of methods based on the transitive closure of a matrix was first considered by Agrawal and Jagadish in [4]. An important observation, reported in [3], is that the matrix elements can be processed in *any* order which satisfies the following constraints:

1. In any row i, an element a_{ik} is processed before a_{ij} iff $k < j$.
2. For any element a_{ij}, processing of a_{jk} precedes a_{ij} iff $k < j$.

The first constraint requires that all the elements to the left of a_{ij} and on the same row are processed before it, and the second constraint requires that all elements on the row with the same number as the column of a_{ij}, and on the left of a_{ij} are processed before it.

In [4] two algorithms are developed in order to achieve a parallel computation that satisfies these two constraints. The algorithms are presented for the reachability problem, but can easily be adapted to solve general path problems. Experimental results presented in [4] refer to computation of the bill-of-material problem.

The basic idea is to partition rows of the adjacency matrix among processors so that each processor owns a contiguous set of nodes, by storing for each node the whole successor list of that node. In order to satisfy the precedence

```
for q := 1 to m do
    begin
    if p = q then
            begin
            (* computation of lower triangle *)
            for i := b_p to e_p
                    begin
                    for j := b_p to i - 1 process a_ij;
                    copy_i := successors_i;
                    show (all, copy_i);
                    end
            (* computation of upper triangle *)
            for i := b_p to e_p
                begin
                for j := i + 1 to e_p process a_ij;
                end
            end
            else
            begin
            (* computation of square *)
            for j := b_q to e_q
                begin
                remote-get (copy_j);
                for i := b_p to e_p process a_ij;
                end
            end
    end
```

Figure 20. The direct algorithm by Agrawal and Jagadish.

constraints, some amount of synchronization and communication is required among processors. Two primitives are used:

- **remote-get.** A *remote-get* is executed by a processor to access a data item not available at the processor itself. The operation is blocked if the data is unavailable.
- **show.** A *show* operation is executed by a processor to make a piece of data available to other processors. A processor may not gain access to remote data unless it has been shown by its owner.

Processors are numbered from 1 to m, processor p owns the pth partition of rows of the matrix, and b_p and e_p denote the first and last row of the pth partition. The first algorithm proposed in [4] is shown in Figure 20, where the program executed at each processor p is listed.

Let us examine Figure 20 in detail. For each value of q, only one processor executes the *if* part and all other processors execute the *else* part. Notice that

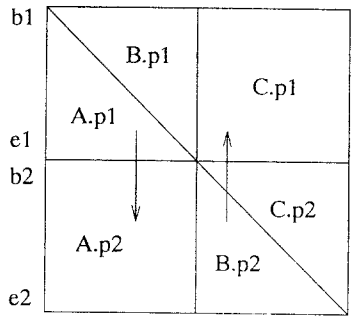

Figure 21. Direct algorithm.

processors do not have to synchronize in the outer *for* loop, since the primitive *remote-get* is blocking and performs synchronization when necessary. When a processor executes the *if* part it does not need access to remote data. It first processes elements below the diagonal row-by-row, sending the result at the end of each row, and then it processes elements above the diagonal. When a processor executes the *else* part, it processes elements in column order, as soon as the rows needed are available.

Because direct algorithms execute loops on tuples and use a matrix-based representation, they cannot be represented with the graphic formalism used in the iterative methods presented so far. A graphic representation of parallel direct methods operating on a partitioned matrix is shown in Figure 21 for the case of two processors. The upper half of the matrix is computed on processor 1, the lower half of the matrix is computed on processor 2. The solid arrows indicate the data flow and implicit synchronization.

The elementary operation of these algorithms is merging successor lists corresponding to the non-null entries in a given successor list. The flow of execution is as follows: On processor p_1, triangle $A.p_1$ is computed first, then triangle $B.p_1$ is computed, and finally, square $C.p_1$ is computed. On processor p_2 square $A.p_2$ is computed first, then triangle $B.p_2$ is computed, and finally, triangle $C.p_2$ is computed. Synchronization is due to the fact that results of $A.p_1$ are necessary for the computation of $A.p_2$; similarly, the results of $B.p_2$ are necessary for the computation of $C.p_1$.

A variant to this algorithm, also proposed in [4], attempts to avoid the time spent in waiting for available rows. In this case, instead of assigning contiguous successor lists to the processors, lists are assigned in a round-robin fashion. When a particular element cannot be processed, the algorithm attempts to go on processing the next element in its partition, instead of blocking; an interrupt is set up in order to detect the availability of rows left behind. Results in [4] show that this variant usually outperforms the first algorithm.

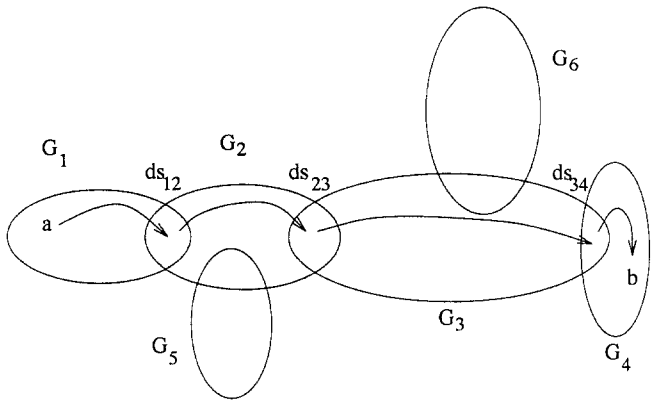

Figure 22. The disconnection set approach.

9. Semantic fragmentation

The basic idea that underlies the disconnection set approach presented by Houtsma, Apers, and Ceri in [27, 28] is best illustrated by an example. Consider a railway network connecting cities in Europe, and a question about the shortest connection between Amsterdam and Milan. Assume that data are naturally fragmented by state (e.g., Holland, Germany, and Italy). Also assume that the border points between states are relatively few. The above 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. All these queries have the same structure; they apply only to a fragment of the database, and can be executed in parallel. Postprocessing is required to assemble the shortest path between the initial and final city, given all shortest paths produced within one fragment. The approach is sketched in Figure 22.

We assume as usual that the base relation R stores the connection information. 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 and the properties of their connections; these properties depend on the particular path problem considered. For instance, for the shortest path problem it is

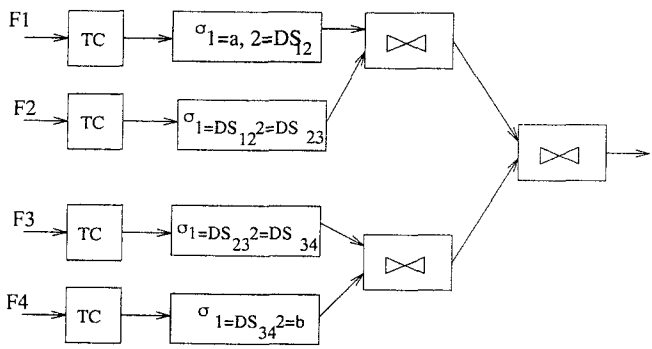

Figure 23. Example of query with the disconnection set approach.

required to precompute the shortest path among any two cities in the border between two fragments. Complementary information about the disconnection set DS_{ij} is stored at both sites storing the fragments R_i and R_j.

An important, but not strictly necessary, property of a fragmentation is to be *loosely connected:* this corresponds to having an acyclic graph G' of components G_i. Formally, $G' = \langle N, E \rangle$ 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 in G there is only one chain of fragments G_i 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 itself) such property does not hold.

In [27] 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[6]. 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, including 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 a chain of length n, query processing is performed in parallel at each computer. Each subquery determines independently a shortest path; 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. 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 Figure 23.

The disconnection set approach is 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 speedup 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.

For evaluating the recursive subquery on a fragment, any suitable single-processor algorithm may be chosen; it is even 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 preprocessing required for building the complementary information and to the careful treatment of updates. Complementary information is different for each type of path problem; in [27, 28] 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.

The problem of designing a fragmentation given an arbitrary graph is described in [29], where several algorithms are presented. Each algorithm fragments a graph according to a different criterion. In [31], Houtsma, Wilschut, and Flokstra describe the implementation of the disconnection set approach on the *PRISMA* database machine. They also give some first performance results, indicating that for some graphs even superlinear speedup is achieved.

A generalization of the disconnection set approach, called *Parallel Hierarchical Evaluation*, is described in [30]. One fragment is designated as high-speed fragment; it contains connections corresponding to *high-speed* transports. The fragmentation is such that the shortest path among any two pairs of nodes is found by considering the fragments where they reside and the high-speed fragments only; therefore, any query can be answered by three processors executing in parallel. This approach mimics real-life transport problems, such as inter-city trains or motor highways, where travels across a large country are done by connecting to the high-speed network from both the city of departure and of arrival.

10. Parallelism in the logical framework

In this section we focus on the parallelization of general Datalog programs.

10.1. General framework

A Datalog program is a finite set of rules whose predicate symbols are divided into two disjoint subsets: *base* predicates and *derived* predicates. Base predicates cannot occur in the head of any rule in a Datalog program. A Datalog query is any goal on a single predicate, typically a derived one[7]. The approaches described in the literature to distributed computation of Datalog queries can broadly be classified in two classes:

- *Program-oriented methods.* These methods generate various versions of the original logic program and assign each of them to a different processor. In some of the approaches relations are partitioned among processors; in these cases, similarities can be found with hash-based methods (see Section 10.2.3).
- *Rule-oriented methods.* These methods assign the execution of each individual rule to a distinct processor; similarities can be found with methods that assign algebraic operations to processors (see Section 10.3.2). The database is either centralized or distributed, but not partitioned.

10.2. Program-oriented methods

Program-oriented methods rewrite the original logic program into several versions, and assign each of them to a different processor. We start by presenting the general schema proposed by Ganguly, Silberschatz, and Tsur [24], which can be used to compute any Datalog program (including non-linear programs and programs with more than one recursive rule). More specialized approaches will be introduced in subsequent subsections.

10.2.1. General queries. Let M be a Datalog program whose rules are numbered from 1 to n, in some order. For each rule R_i in M, let $v(R_i)$ be any sequence of variables, all of which appear in the rule R_i. This sequence is referenced as the *discriminating sequence* for the rule R_i. Let $\mathcal{P}$ be a finite set of processors, (e.g., $\{1, 2, \ldots, n\}$) on which the program is to be executed. Then, the hash function h_i is defined as follows:

$$h_i : \text{ set of ground instances of } v(R_i) \rightarrow \mathcal{P}$$

h_i is called the *discriminating function* of R_i.

We now derive from M a set of Datalog programs to be executed at the various processors. The parallel execution of this derived set of Datalog programs is

equivalent to the sequential execution of M. Let M_i denote the program to be executed at processor i. It consists of the following four execution steps.

1. **Processing.** Let $A :- B, \ldots, C$ be a rule in M, with discriminating sequence $v(R)$ and a discriminating function h. Then include the following rule in M_i:

   $$A^i_{\text{out}} :- B^i_{\text{in}}, \ldots, C^i_{\text{in}}, \; h(v(R)) = i$$

   The interpretation of the new predicate A^i_{out} is the set of all the A-tuples generated at processor i. The interpretation of predicates $B^i_{\text{in}}, \ldots, C^i_{\text{in}}$ is the set of all B-tuples, C-tuples, etc. that are input to processor i at some point in the execution.
2. **Sending.** Let R be a rule in M, with discriminating sequence $v(R)$ and discriminating function h. For every recursive atom C appearing in R and every $j \in \mathcal{P}$, include the following rule in M_i:

   $$C_{ij} :- C^i_{\text{out}}, \; h(v(R)) = j$$

   The interpretation of C_{ij} is the required communication from processor i to processor j.
3. **Receiving.** Let $\vec{W}$ be a sequence of appropriate length of new, distinct variables. For every recursive predicate T appearing in the program M and every $j \in \mathcal{P}$, introduce the following rule in M_i:

   $$T^i_{\text{in}}(\vec{W}) :- T_{ij}(\vec{W})$$

   This rule indicates the tuples that are received at processor i.
4. **Final pooling.** For every recursive predicate T, include the following rule in M_i, that is responsible for collecting all the tuples computed for T:

   $$T(\vec{W}) :- T^i_{\text{out}}(\vec{W})$$

The abstract architecture on which the parallel program is executed assumes that each processor $i \in \mathcal{P}$ may communicate with every other processor $j \in \mathcal{P}$. The parallel execution proceeds with each processor i evaluating the Datalog program M_i using semi-naive evaluation. The predicates C_{ij}, for $i, j \in \mathcal{P}$, represent the channel ij in the abstract architecture. Hence, addition of tuples to the predicate C_{ij} should be interpreted as processor i sending the tuples to processor j, along channel ij.

The general structure of the parallel execution at each processor is

repeat
  Evaluate processing rules
  Evaluate sending rules
  Evaluate receiving rules
until *Termination*

where *Termination* is the condition that all processors are idle and all channels are empty.

The choice of the discriminant sequence, that is the sequence of variables whom the discriminating function h is applied to, must be restricted in order to obtain effective parallelism. If the variables appearing in $v(R)$ do not appear in any of the atoms in the body, then each processor has to compute joins over the entire relations, since selections over the value of the discriminating function cannot be pushed into joins. Thus, for the remainder of the section we assume that all variables appearing in the discriminating sequence for the recursive rules must also appear in at least one atom in the body of the recursive rules.

The base relations are distributed among the processors in the following way. Suppose R is a rule with discriminating sequence $v(R)$ and D is the symbol of a base predicate occurring in R. If the variables appearing in $v(R)$ do not appear in D, then D is shared or replicated among the processors. Otherwise, the fragment of D accessed by processor i is denoted by D^i_{in} and is defined by:

$$D^i_{\text{in}} :- D,\ h(v(R)) = i$$

This fragmentation phase can take place before starting the processing of rules (actually, it may exist before query execution).

10.2.2. Data reduction paradigm for linear rules. A similar approach that applies to general Datalog programs, called *data reduction paradigm*, is presented by Wolfson and Ozeri in [66]. We omit to describe general queries, and concentrate on linear rules. In this case, the data reduction paradigm can be specialized in order to have parallelization strategies that do not require communication, though they have some redundancy.

The schema presented by Wolfson in [67] can be applied to any linear sirup; we consider our standard example (3), but we keep base relations R_1 and R_2 distinct:

$$P(X, Y) :- R_1(X, Y)$$
$$P(X, Y) :- R_2(X, Z), P(Z, Y)$$

Let $v(E)$ be a discriminating sequence for the nonrecursive rule ($v(E)$ in the example is either (X), or (Y), or (X, Y), or (Y, X)). Let $\mathcal{P}$ be a set of processors and h' a discriminating function:

$$h' :\ \text{set of ground instances of } v(E) \to \mathcal{P}$$

The program to be executed at processor i consists of the following three execution steps.

1. **Initialization.** A new predicate P^i is defined whose interpretation is the fragment of R_1 initially stored at the processor i:

$$P^i(X, Y) :- R_1(X, Y),\ h'(v(E)) = i$$

2. **Recursive processing**

$$P^i(X, Y) :- R_2(X, Z),\ P^i(Z,\ Y)$$

3. **Final pooling**

$$T(X,\ Y) :- T^i(X, Y)$$

In this schema, no communication is necessary during the computation, but the same tuple may be generated in the parallel execution more times than in the sequential semi-naive computation. Hence, this approach introduces some redundancy. A precise characterization of the linear programs that can be computed in parallel with neither communication nor redundancy has been given by Ganguly, Silberschatz, and Tsur [24].

10.2.3. Relationship between program-oriented methods for Datalog and hash-based methods for transitive closure. The approaches presented in Section 7 can be interpreted as specific strategies allowed by the more general methods available for linear Datalog rules. Consider again the program

$$P(X, Y) :- R_1(X, Y)$$
$$P(X,\ Y) :- R_2(X, Z),\ P(Z,\ Y)$$

Let $v(R_2)$ and $v(R_1)$ be the discriminating sequences for the recursive and nonrecursive rules, and let discriminating functions h_1 and h_2 be defined as follows:

h_1 : set of ground instances of $v(R_1) \rightarrow \mathcal{P}$
h_2 : set of ground instances of $v(R_2) \rightarrow \mathcal{P}$

Consider the following parallelization schema (as an instance of the general method shown in Section 10.2.1):

1. **Initialization**

$$R_1^i(X, Y) :- R_1(X, Y),\ h_1(v(R_1)) = i$$
$$R_2^i(X, Y) :- R_2(X, Y),\ h_2(v(R_2)) = i$$

2. **Processing**

$$P_{\text{out}}^i(X, Y) :- R_1(X, Y),\ h_1(v(R_1)) = i$$
$$P_{\text{out}}^i(X, Y) :- R_2(X, Z),\ P_{\text{in}}^i(Z, Y),\ h_2(v(R_2)) = i$$

3. **Sending**[8]

$$P_{ij}(X, Y) :- P_{\text{out}}^i(X, Y),\ h_2(v(R_2)) = j$$

4. **Receiving**

$$P^i_{\text{in}}(X, Y) :- P_{ji}(X, Y)$$

5. **Final pooling**

$$P(X, Y) :- P^i_{\text{out}}(X, Y)$$

Then, the method by Valduriez and Khosafian, presented in Section 7.1, corresponds to choosing $v(R_1) = \langle X, Y \rangle$, $v(R_2) = \langle X, Z \rangle$, and $h = h_1 = h_2$ being defined as follows:

$$h(a, b) = i \Leftrightarrow (a, b) \in R^i$$

This choice has two properties:

- The execution of each local program only requires to access its given fragment R^i of R.
- Since R^j is not available at processor i, the discriminating function $h(X, Z)$ in the sending rule cannot be computed at processor i: hence, all the tuples in R^i_{out} have to be transmitted to every other processor j.

Also the approach of Cheiney and de Maindreville described in Section 7.2 is a special case of this schema. Suppose that the base relation R is hashed on the second attribute and fragmented and then hashed on the first attribute; fragmentation is done according to the first hashing. We obtain k^2 fragments R^{ij} for $0 \leq i, j \leq k$, and the number of processors in $\mathcal{P}$ must also be k^2. Let us indicate by $i \times j$ the index of the processor having R^{ij} as its local fragment. Let $v(R_1) = \langle X \rangle$, $v(R_2) = \langle Z \rangle$, and $h' = h$ be defined as follows:

$$h(a) = i \Leftrightarrow \forall b \; \exists (a, b) \in R^{ij}$$

The sending rules now become

$$P_{(i\times j)(k\times l)}(X, Y) :- P^{i\times j}_{\text{out}}(X, Y), \; h(X) = l.$$

Given that $R^{i\times j}$ is doubly hashed, $P^{i\times j}_{\text{out}}$ is still hashed on the first attribute, and therefore the condition $h(X) = i$ holds. Tuples are therefore transmitted only from processors $P^{i\times j}$ to processors $P^{k\times i}$. The amount of communication is therefore reduced with respect to the previous approach, but all its advantages are retained. In particular, each processor needs only to access one fragment of the base relation R.

Finally, also the method proposed by Agrawal and Jagadish and described in Section 7.3 has its logical counterpart, which is the specialized data reduction method for linear rules proposed by Wolfson and outlined in Section 10.2. Figure 19 shows precisely the Wolfson approach with $R_1 = R_2 = R$. The vector $v(e)$ is simply (Y), the second variable of the nonrecursive rule, corresponding

to the second column of R; the discriminating function builds k fragments R_i, $1 \leq i \leq k$. In particular, note that this method distributes both one fragment and a full copy of R to each processor, while approaches in Figures 17 and 18 use hashing also on the second copy of R.

10.3. Rule-oriented methods

In these methods, processor assignment depends only on the structure of the rules. The evaluation of a query is decomposed into two parts.

1. The logic program is compiled into an internal structure (called rule/goal graph in [63] or derivation tree in [32]) that essentially reflects the properties of intensional predicates, and highlights the order in which rules should be executed so that efficient database access is performed. Details about building rule-goal graphs can be found in [58].
2. Then, each rule is assigned to a processor. Query execution, for a particular goal structure and internal program representation, produces the answer to the query. Each rule processor stores intermediate relations consisting of all the tuples that are produced for that rule. Messages containing the produced tuples are exchanged among rule processors. Query execution is halted when termination conditions are met; these, as in the previous section, are reached when each intermediate relation has reached a fixpoint and all tuples have been transferred along the channels between processors.

10.3.1. Distributed rule evaluation on rule-goal graphs. Van Gelder [63] describes a method for building a network of cooperating processes for a given rule/goal graph. During the dynamic phase there is an initialization part, in which the specifications deduced by inspecting the goal are exchanged among processors; each processor informs its neighbor processes about its needs. Then, messages are exchanged among processors. For recursive queries, the network of communicating processes is cyclic.

Van Gelder describes a termination algorithm that is structured in two phases. It uses one of the spanning trees in the network of communicating processes. Several termination messages are sent from a termination coordinator process, at the root of the spanning tree, toward the leaves of the spanning tree itself. When termination messages are received by the leaves of the spanning tree, they are returned to the termination coordinator. This process is iterated twice; the double iteration is required by this schema to ensure that all communication channels are empty and that all processors are idle.

The approaches of Hulin [32] and Shao et al. [55], are essentially a follow-up of [63]. We give a brief overview of Hulin's approach by an example. Consider the *same generation* program, containing a nonrecursive and a stable linear

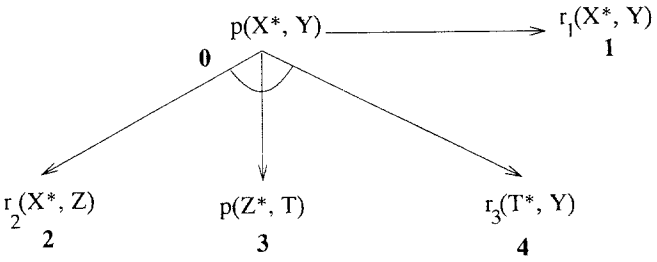

Figure 24. Derivation tree.

recursive rule:

$$P(x, y) \leftarrow R_1(x, y)$$
$$P(x, y) \leftarrow R_2(x, x1), P(x1, y1), R_3(y1, y)$$

Let the goal be? – $P(a, Y)$.

A *query scheme* of a predicate P is any atomic formula $P(x_1, \ldots, x_n)$ without constants and whose variables are divided between *entry variables* (denoted by x_i^*) and *exit variables* (denoted simply by x_i). The query scheme associated with the goal of our example is therefore $P(x^*, y)$.

During *query compilation*, a *derivation tree* is built by recursively splitting each query scheme into subquery schemes, one for each deduction rule defining the query scheme predicate. When several equivalent query schemas are present in the derivation tree, only one of them is explicitly decomposed; the subquery order in a decomposition is imposed by a selection function. The derivation tree for the same generation example is shown in Figure 24. Note that exit variables of a query schema become entry variables for query schemas that follow it in the decomposition order.

In the derivation tree there may be several different nodes labeled with equivalent query schemes. The first one encountered during a depth-first traversal of the tree is called *archetype* node. In the above example, nodes 0, 1, 2, 4 are archetype nodes. In the subsequent *query evaluation* phase, one process **Evaluate**(n) is created for each archetype node n. The same generation example will therefore be solved by four cooperating processes. More precisely, to each node n is associated

- A process **Evaluate**(n) that computes and propagates the answers to *active queries* at n.
- A private memory **Memory**(n) that contains the set of *active queries* at n.
- Two buffers **Request**(n) and **Answer**(n) to store messages sent by other processes. Requests are pairs composed by an entry value and a context for a node whose archetype is n. Answers are composed by a set of variable/value pairs.

During the evaluation process, information units are called *active queries* and are associated with each node to remember queries previously requested at the node, their partial solution, and the contexts in which the requests were issued.

The algorithm can be sketched as follows: request messages are stored as active queries in **Memory**(n); answer messages are used to fill the set of exit variables of an active query. To compute an active query, a node has three possibilities:

- Retrieving tuples from the extensional database, if it is a base node
- Using values previously computed and stored in **Memory**(n)
- Issuing a request to some other nodes in order to solve a subquery

In our example, in order to evaluate the query $? - P(a, Y)$ a request is issued to the root node n_0. In **Memory**(n_0) will therefore be stored the active query $(\{X/a\}, \{\ \}, \{\langle n_0 \rangle\})$. The node n_0 will send requests $(\{X/a\}, \langle n_1 \rangle)$ and $(\{X/a\}, \langle n_2 \rangle)$ to nodes n_1 and n_2 respectively. This process is continued until termination; at the end of the computation, the set of answers will be present in **Memory**(n_0).

10.3.2. Relationship between rule-oriented methods for Datalog and operation-to-processor mappings. Rule-oriented methods can be interpreted as a particular operation-to-processor mapping; indeed, computing a rule corresponds to evaluating join operations predicates in the right side. Results produced by the computation are progressively accumulated in local memory through unions. With respect to the mappings described in Section 6, mappings induced by the structure of rules are much more general, and processors must be able of performing several functions (storing tuples, making requests, responding to them); however, these functions are regular, and all processors can be programmed in the same way, to serve a generic rule rather than a specific operation.

11. Impact of initial data distribution upon performance

In this section, we focus on the method by Valduriez and Khoshafian, described in Section 7.1, and investigate the impact of initial data distribution upon performance. We assume base tables corresponding to graphs of known structure: lists, regular trees, regular directed acyclic graphs, and regular cyclic graphs. The choice of one method is arbitrary, but also rather unessential; indeed, we are uniquely interested in showing that performance is *significantly* affected by the above factors.

Performance is evaluated in terms of

Communication the total number of tuples transmitted from any processor to any other processor.

Speedup the ratio between the amount of computation needed by one processor

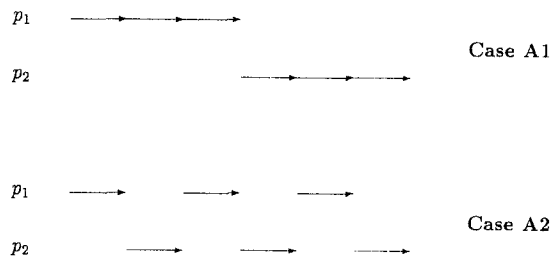

Figure 25. Data distributions considered for lists.

for completing the transitive closure and the amount of computation assigned to the most heavily loaded processor when we use p processors in parallel. We approximate computation cost for joins with the size of its result. Note that with this definition the ideal speedup factor is p.

11.1. Initial data distribution

We consider graphs representing lists, trees, DAGs, and cyclic graphs.

List We consider two possible distributions, shown in Figure 25. In case **A1**, sublists formed by consecutive edges are assigned to each fragment. In case **A2**, the edges of the list are distributed to fragments with a round-robin schema.

Tree We consider two possible distributions, shown in Figure 26. In case **B1**, entire subtrees are assigned to each fragment. In case **B2**, all edges originating from children at the same depth level are assigned to the same processor; levels are assigned to processors with a round-robin schema.

Dag We use a simple DAG, reproducing a sequence of triangles; we consider two possible distributions, shown in Figure 27. In case **C1**, each fragment consists of a consecutive list of triangles; this case is similar to **A1**. In case **C2**, the triangles are distributed to the fragments with a round-robin schema; this case is similar to **A2**.

Cyclic We use a simple cyclic schema, reproducing a sequence of squares; we consider two distributions, shown in Figure 28. In case **D1**, each fragment contains a consecutive list of squares (similarly to **A1**); in case **D2**, the squares are distributed to fragments with a round-robin schema (similarly to **A1**).

Let n denote the number of tuples in R, $|T|$ denote the cardinality of the transitive closure of R, and p denote the number of processors; we assume n to be a multiple of p. We now derive a formula for the communication and speedup for each of the initial data distributions.

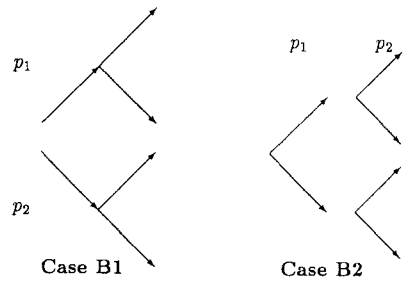

Figure 26. Data distributions considered for trees.

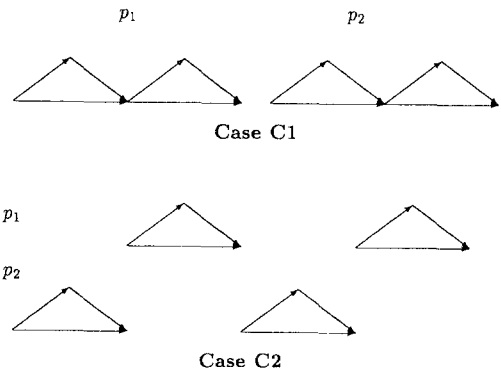

Figure 27. Data distributions considered for DAGs.

11.1.1. A1

Communication. The last processor generates paths from length 1 up to n/p (one of each length) and sends them to its predecessor. This predecessor thus generates paths from length 1 up to $2n/p$ (one of each length) and sends them in turn to its predecessor. Therefore, the total communication cost is

$$np^{-1} + 2np^{-1} + 3np^{-1} + \cdots + (p-1)np^{-1} = np^{-1}\sum_{i=1}^{p-1} i = \frac{n(p-1)}{2}$$

The above number can be compared with the total size of the transitive closure, $|T|$:

$$n + (n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = |T|$$

Then, the total communication cost is equal to $|T| \times ((p-1)/(n+1))$.

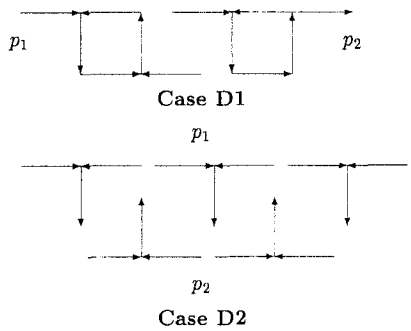

Figure 28. Data distribution considered for cyclic databases.

Speedup. The workload of processors in this case in uneven; the first processor performs much more work than the last one. Let us consider the last processor (pth) first. Its first iteration requires a join of two operands, each of size n/p (the entire fragment). In the subsequent iterations, the size of one of the two operand reduces by 1, while the other operand remains of the same size. The amount of work at each iteration thus gradually reduces from joining a relation of size n/p with one of size n/p, to joining a relation of size n/p with one of size 1.

Let us consider now the processor $p-1$. In its first n/p iterations, it always computes a join between two relations of size n/p; this happens because at each iteration one tuple is transmitted and one tuple is received. In the subsequent n/p iterations, the work reduces as illustrated for the last processor. Processor $p-1$ terminates after $2n/p$ iterations. The ratio between the workload of the first processor (the one determining the response time) and the last one (which terminates first) is thus $2(p-1)+1$. If there were only one processor, its total workload would be p^2 times the workload of the first computer. Therefore, the speedup is $p^2/(2(p-1)+1)$.

11.1.2. A2

Communication. At each iteration, a processor must send all the tuples generated in the previous iteration to another processor. Hence, first all n tuples representing paths of length 1 are sent, then all $n-1$ tuples representing paths of length 2 are sent, etc. Therefore, the total communication is the same as the size of the transitive closure $|T|$.
Speedup. The workload is evenly distributed over the processors, and the speedup is well approximated with p.

11.1.3. B1. We assume that each node has the same number of descendents; this number is equal to p, the number of processors.

Communication. No communication is needed.
Speedup. The workload is evenly spread over the processors; the speedup is p.

11.1.4. B2

Communication. By applying the same reasoning as in case **A2**, the entire transitive closure $|T|$ needs to be transmitted.
Speedup. We assume that the number of processors p equals the depth of the tree, and that there is a constant fan-out f. Processor p performs no join, and transmits f^p tuples to processor $p-1$; this processor joins the incoming tuples producing f^p tuples, and transmits $f^{p-1} + f^p$ tuples to processor $p-2$. Processor $p-2$ joins the incoming tuples, producing $f^{p-1} + f^p$ tuples, and so on. The most heavily loaded processor is the first one, it computes $\sum_{i=2}^{p} f^i$ tuples.

From the observations above we may see that the total work load is $\sum_{i=2}^{p}(i-1)f^i$. The speedup is therefore $\sum_{i=2}^{p}(i-1)f^i / \sum_{i=2}^{p} f^i$. It is easily seen that speedup is smaller than $p-1$. Using a numerical analysis, we may notice that speedup is larger than $p-2$.

11.1.5. C1 and C2. For brevity, we combine cases C1 and C2. Recall that in DAGs there exist multiple paths (of different length) between points. In a hash-based approach, this leads to the generation of many redundant tuples that have to be removed. Let $\vec{T}$ stand for the number of independent paths in the transitive closure of the graph corresponding to R; in general, $|\vec{T}| >> |T|$.

Communication. In case **C2** communication is $2/3\ |\vec{T}|$; two out of three paths start from the first node and will be communicated. In case **C1** communication occurs only from the first nodes of each fragment; there are $p-1$ such nodes, instead of $n/3$ as in case **C2**. Thus, communication is: $2(p-1)/n\ |\vec{T}|$.
Speedup. The speedup behavior of cases **C1** and **C2** is the same as that of cases **A1** and **A2**, respectively.

11.1.6. D1 and D2. With cycles, difference operations must be computed to detect termination; the cost of evaluating such difference operations is not considered here.

Communication. In case **D2**, a total of $|\vec{T}/2|$ paths needs to be communicated; indeed, from each node there depart exactly n independent paths (this may be observed in Figure 28 by considering that each edge contributes exactly one path), however only half of the nodes originate communications. In case **D1** communication occurs only from the first nodes of each fragment; therefore, $|\vec{T}|/p$ independent paths need to be transmitted.
Speedup. Due to the symmetry of cases **D1** and **D2**, where the amount of

Table 1. Characteristics of hashing approaches for example distributions.

Initial data distribution	Communication	Speedup
A1	$\|T\| \times ((p-1)/(n+1))$	$p^2/(2(p-1)+1)$
A2	$\|T\|$	p
B1	0	p
B2	$\|T\|$	$\approx (p-1)$
C1	$2(p-1)/n \ \|\vec{T}\|$	$p^2/(2(p-1)+1)$
C2	$2/3 \ \|\vec{T}\|$	p
D1	$\|\vec{T}\|/p$	p
D2	$\|\vec{T}\|/2$	p

computation at each processor is exactly the same, the speedup is p in both cases.

11.1.7. Comparison. The communication and speedup computed for the eight initial data distributions are summarized in Table 1. Speedup approximates p in six cases and $p/2$ in two cases. Communication is null only in one case. When no multiple paths exist among nodes (cases A and B), communication approximates $|T|$ (the cardinality of the transitive closure) in two cases out of four; and approximates $|T|$ multiplied by a factor p/n in one case. Whenever there can be multiple paths in the graph, communication increases significantly. It is proportional to $|\vec{T}|$ (the number of independent paths in the transitive closure) in two cases; and is otherwise approximated by $|\vec{T}|$ multiplied by a factor $1/p$ or by a factor $2p/n$.

Data distributions of cases B1 and D1 dominate the distributions of cases B2 and D2, because corresponding executions have better communication and speedup. Indeed, they correspond to ideal data fragmentation: they would be produced by applying semantic fragmentation principles (the disconnection set is the root node in case B1 and the set of the two nodes separating $p1$ from $p2$ in case D1).

The comparison between A1, C1 and A2, C2 is more difficult, since the former cases have worse speedup but better communication. The implication is that designing an ideal data distribution is not obvious even in the simple case of regular lists and DAGs.

12. Conclusions

This paper has presented an overview of techniques for parallel evaluation of recursive queries, reporting recent research results; most of our references have been written in the last five years. While the theory of parallel recursion can be considered sufficiently solid and stable, its practical applicability to solve

real problems, in the context of real prototypes and systems, still needs to be assessed. Indeed, most of the published papers have demonstrated the merits of the proposed approaches by means of analytical models or through simulations, but there is a general lack of experience of these techniques in concrete environments (see, e.g., [31] for an exception). In the near future, with the spreading of multiprocessor architectures and the growth of their application to intraquery parallelism, this problem will become more and more relevant, and the systems will naturally evolve their ability of computing parallel joins into the ability of computing recursive queries in parallel.

Acknowledgments

The authors wish to thank anonymous referees for stimulating several additions and improvements to preliminary versions of this paper. This research is partially supported by the LOGIDATA+ Project of the National Research Council of Italy. Stefano Ceri is partially supported by Esprit Project P6333, IDEA and the research of Maurice Houtsma has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences.

Notes

1. Ideas similar to seminaive optimization were used earlier in the context of loop optimization [23, 45].
2. The *enable* signal is not shown in the figures, since it is required starting at iteration 4.
3. Assume the P_{bf} query and compare the first line of joins and unions for P_{bf} terms with the naive evaluation shown in Figure 10: they are the same. The other blocks of this architecture do not produce useful tuples.
4. The *enable* signal is not shown in the figures, it is required starting at iteration 2.
5. The *enable* signal is not shown in the figures, since it is required starting at iteration 3.
6. Note that the shortest path might include nodes *outside* the chain, however their contribution is precomputed in the complementary information.
7. Kanellakis, Van Gelder and Ullman [38, 59] have investigated Datalog programs and indicated whether they belong to NC (a program in NC can be evaluated in polylogarithmic time given a polynomial number of processors). This characterization is mainly of theoretical interest, since a polynomial number of processors in the size of the database is too high for conventional database applications.
8. Duplicate tuples generated by the same processor may be detected by a set difference operation; set difference can be explicitly added to this logic program through a *locally stratified negation*.

References

1. R. Agrawal, S. Dar, and H.V. Jagadish, "Direct transitive closure algorithms: Design and performance evaluation," *ACM Trans. Database Systems*, vol. 15, no. 3, pp. 427–458, 1990.
2. R. Agrawal and P. Devanbu, "Moving selections into linear least fixpoint queries," *IEEE Trans. Knowledge Data Engg.*, vol. 1, no. 4, pp. 424–432, 1989.

3. R. Agrawal and H.V. Jagadish, "Direct algorithms for computing the transitive closure of database relations," in *Proc. 13th Int. Conf. Very Large Data Bases*, Brighton, 1987, pp. 255–266.
4. R. Agrawal and H.V. Jagadish, "Multiprocessor transitive closure algorithms," in *Proc. Int. Symp. Databases in Parallel and Distributed Systems*, Austin, TX, 1988.
5. A.V. Aho, and J.D. Ullman, "Universality of data retrieval languages," *Sixth ACM Symp. Principles of Programming Languages*, San Antonio, 1979.
6. P.M.G. Apers, A. Hevner, and B. Yao, "Optimization algorithms for distributed queries," *IEEE Trans. Software Engg.*, vol. 9, no. 1, 1983.
7. P.M.G. Apers, M.A.W. Houtsma, and F. Brandse, "Processing recursive queries in relational algebra," in *Data and Knowledge (DS-2), Proc. 2nd IFIP 2.6 Working Conf. Database Semantics, Albufeira, Portugal, 1986*, R.A. Meersman and A.C. Sernadas (Eds.), North-Holland, 1988, pp. 17–39.
8. P.M.G. Apers, M.L. Kersten, and H. Oerlemans, "PRISMA database machine: a distributed main-memory approach," in *Advances in Database Technology, Proc. Int. Conf. Extending Database Technology* (EDBT), 1988, pp. 590–593.
9. I. Balbin and K. Ramamohanarao, "A generalization of the differential approach to recursive query evaluation," *J. Logic Program.*, vol. 4, no. 3, pp. 259–262, 1987.
10. F. Bancilhon, "Naive evaluation of recursively defined relations," Technical Report DB-004-85, MCC, Austin, TX, 1985.
11. F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, "Magic sets and other strange ways to implement logic programs," *Proc. ACM SIGMOD-SIGACT Symp. Principles of Database Systems*, Cambridge, MA, 1986.
12. F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing," *Proc. ACM SIGMOD Conf.* Washington DC, 1986.
13. S. Ceri, G. Gottlob, and L. Lavazza, "Translation and optimization of logic queries: the algebraic approach," in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, 1986, pp. 395–403.
14. S. Ceri, G. Gottlob, and L. Tanca, *Logic programming and Databases*, Springer-Verlag, 1990.
15. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.
16. S. Ceri and L. Tanca, "Optimization of systems of algebraic equations for evaluating Datalog queries," in *Proc. 13th Int. Conf. Very Large Data Bases*, Brighton, 1987, pp. 31–42.
17. J.P. Cheiney and C. De Maindreville, "A parallel strategy for transitive closure using double hash-based clustering," in *Proc. 16th Int. Conf. Very Large Data Base*, Brisbane, 1990.
18. G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data Placement in Bubba," in *Proc ACM-SIGMOD Conf.*, 1988, pp. 99–108.
19. M.P. Consens and A.O. Mendelzon, "GraphLog: a visual formalism for real life recursion," in *Proc. ACM SIGMOD-SIGACT Symp. Principles of Database Systems*, Nashville, 1990, pp. 404–416.
20. I.F. Cruz and T.S. Norvell, "Aggregative closure: an extension of transitive closure," in *Proc. 5th Inf. Conf. Data Engineering*, Los Angeles, 1989, pp. 384–391.
21. S. Dar, R. Agrawal, H.V. Jagadish, "Optimization of generalized transitive closure queries," in *Proc. 7th Int. Conf. Data Engineering*, Kobe, 1991, pp. 345–354.
22. D.J. De Witt, S. Ghandeharizadeh, and D. Schneider, "A performance analysis of the Gamma database machine," in *Proc. ACM-SIGMOD Conf.*, 1988, pp. 350–360.
23. A.C. Fong and J.D. Ullman "Induction variables in very high level languages," in *Proc. 3rd ACM Symp. Principles of Programming Languages*, Atlanta, pp. 104–112.
24. S. Ganguly, A. Silberschatz, and S. Tsur, "A framework for the parallel processing of Datalog queries," in *Proc. ACM-SIGMOD Conf.*, Atlantic City, 1990.
25. N. Goodman, P.A. Bernstein, E. Wong, C.L. Reeve, and J.B. Rothnie, "Query processing in SDD-1—A system for distributed databases," *ACM Trans. Database Sys.*, vol. 6, no. 4, 1981.
26. M.A.W. Houtsma and P.M.G. Apers, "Algebraic optimization of recursive queries," *Data Knowledge Engg.* vol. 7, no. 4, pp. 299–325, 1992.
27. M.A.W. Houtsma, P.M.G. Apers, and S. Ceri, "Distributed transitive closure computation: the disconnection set approach," in *Proc. 16th Int. Conf. Very Large Data Bases*, Brisbane, 1990.

28. M.A.W. Houtsma, P.M.G. Apers, and S. Ceri, "Complex transitive closure queries on a fragmented graph," in *Proc. 3rd Int. Conf. Database Theory*, Paris, 1990, Lecture Notes in Comput. Sci., vol. 470, Springer-Verlag, New York.
29. M.A.W. Houtsma, P.M.G. Apers, and G.L.V. Schipper, "Data fragmentation for parallel transitive closure strategies," in *Proc. 9th Int. Conf. Data Engineering*, Vienna, 1993, pp. 447–456.
30. M.A.W. Houtsma, F. Cacace, and S. Ceri, "Parallel hierarchical evaluation of transitive closure queries," in *Proc. 1st Int. Conf. Parallel and Distributed Information Systems*, Miami Beach, 1991, IEEE Computer Science Press.
31. M.A.W. Houtsma, A.N. Wilschut, and J. Flokstra,, "Implementation and performance evaluation of a parallel transitive closure algorithm on PRISMA/DB," in *Proc. 19th Int. Conf. Very Large Data Bases*, Dublin, 1993.
32. G. Hulin, "Parallel processing of recursive queries in distributed architectures," in *Proc. 15th Int. Conf. Very Large Data Bases*, Amsterdam, 1989, pp. 87–96.
33. H.V. Jagadish, R. Agrawal, and L. Ness, "A study of transitive closure as a recursion mechanism" in *Proc. ACM-SIGMOD Conf.*, 1987, pp. 331–344.
34. B. Jiang, "A suitable algorithm for computing partial transitive closures in databases, in *Proc. 6th Int. Conf. Data Engineering*, Los Angeles, 1990, pp. 264–271.
35. Y.E. Ioannidis, "On the computation of the transitive closure of relational operators," in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, 1986, pp. 403–411.
36. Y.E. Ioannidis and R. Ramakrishnan, "Efficient Transitive Closure Algorithms," in *Proc. 14th Int. Conf. Very Large Data Bases*, Los Angeles, 1988.
37. Y.E. Ioannidis and E. Wong, "Towards an algebraic theory of recursion" *J. ACM*, vol. 18, no. 2, 1991.
38. P. Kanellakis "Parallel complexity of logic programs." in *Foundations of Logic Programming and Deductive Databases*, Morgan-Kauffman, 1988.
39. M. Kifer and E.L. Lozinskii, "Filtering data flow in deductive databases," in *Proc. 1st Int. Conf. Database Theory*, Rome, 1986.
40. G. Kleinhuis and K.R. Oskam, "Evaluation and simulation of parallel algorithms for the transitive closure operation," M.Sc. thesis, University of Twente, The Netherlands, 1989.
41. I.S. Mumick, S.J. Finkelstein, and H. Pirahesh, "Magic conditions," in *Proc. ACM SIGMOD-SIGACT Symp. Principles of Database Systems*, 1990.
42. I.S. Mumick, S.J. Finkelstein, and H. Pirahesh, "Magic is relevant," in *Proc. ACM SIGMOD Conf.*, Atlantic City, 1990.
43. I.S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The magic of duplicates and aggregates," in *Proc. 16th Int. Conf. Very Large Data Bases*, Brisbane, 1990.
44. S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases*, Computer Science Press, 1989.
45. R. Paige and J.T. Schwartz, "Reduction in strength of high level operations," in *Proc. 4th ACM Symp. Principles of Programming Languages*, Los Angeles, 1977, pp. 58–71.
46. M.J. Quinn and N. Deo, "Parallel graph algorithms," *ACM Comput. Surv.*, vol. 16, no. 3, 1984.
47. R. Ramakrishnan "Magic templates, a spellbinding approach to logical evaluation," in *Proc. Logic Programming Conf.* 1988.
48. R. Ramakrishnan, C. Beeri, and R. Krishnamurty, "Optimizing existential Datalog queries," in *Proc. ACM SIGMOD-SIGACT Symp. Principles of Database Systems*, Austin, TX, 1988.
49. L. Raschid and S.Y.W. Su, "A parallel strategy for evaluating recursive queries," *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, 1986.
50. J. Robinson and S. Lavington, "A transitive closure and magic function machine," in *Proc. 2nd Int. Symp. Databases in Parallel and Distributed Systems*, Dublin, 1990, pp. 44–54.
51. J. Rohmer, R. Lescoeur, and J.M. Kerisit, "The Alexander method: a technique for the processing of recursive axioms in deductive database," in *New Generation Comput.*, vol. 4, Springer-Verlag, 1986.

52. Y. Sagiv "Optimizing Datalog programs" in *Proc. ACM SIGMOD-SIGACT Symp. Principles of Database Systems*, San Diego, 1987.
53. L. Schmitz "An improved transitive closure algorithm," *Computing*, vol. 30, 1983, pp. 359–371.
54. D.A. Schneider and D.J. De Witt, "A performance analysis of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proc. ACM SIGMOD Conf.* Portland, 1989, pp. 110–121.
55. J. Shao, D.A. Bell, and M.E.C. Hull, "An experimental performance study of a pipelining recursive query processing strategy," in *Proc. 2nd Int. Symp. on Databases in Parallel and Distributed Systems*, Dublin, 1990, pp. 30–43.
56. The 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.
57. Teradata Corporation, "Teradata DBC/1012 data base computer concepts and Facilities," release 3.1 edition 1988, Teradata Document C02-0001-05.
58. J.D. Ullman *Principles of Data and Knowledge-Based Systems*, Computer Science Press, 1989.
59. J.D. Ullman and A. Van Gelder, "Parallel complexity of logic programs," Technical report STAN-CS-85-1089, Stanford University.
60. P. Valduriez and H. Boral, "Evaluation of recursive queries using join indices," in *Proc. 1st Int. Conf. Expert Database Systems*, Charleston, 1986; and *Expert Database Systems*, L. Kerschberg (Ed.), Benjamin-Cummings, 1987.
61. P. Valduriez and S. Khoshafian, "Parallel evaluation of the transitive closure of a database relation," *Int. J. Parallel Program.*, vol. 17, no. 1, 1988.
62. P. Valduriez and S. Khoshafian, "Transitive closure of transitively closed relations," in *Proc. 2nd Int. Conf. Expert Database Systems*, Vienna, VA., 1988; and *Expert Database Systems*, L. Kerschberg (Ed.), Benjamin-Cummings, 1988, pp. 377–400.
63. A. Van Gelder, "A message passing framework for logical query evaluation," in *Proc. ACM SIGMOD Conf.*, pp. 155–165.
64. H.S. Warren, "A modification of Warshall's algorithm for the transitive closure of binary relations," *Comm. ACM*, vol. 18, no. 4, pp. 218–220, 1975.
65. S. Warshall, "A theorem on Boolean matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, 1962.
66. O. Wolfson and A. Ozeri, "A new paradigm for parallel and distributed rule-processing," in *Proc. ACM SIGMOD Conf.*, pp. 113–142.
67. O. Wolfson "Sharing the load of logic program evaluation," *Int. Symp. Databases in Parallel and Distributed Systems*, Austin, TX, 1988, pp. 46–55.
68. C. Zaniolo and D. Saccà, "Rule rewriting methods for efficient implementation of Horn logic," MCC technical report DB-084-87, Austin, TX, 1987.