Parallel Processing of Recursive Queries in Distributed Architectures

Guy Hulin

Philips Research Laboratory
2 avenue Van Becelaere, 1170 Brussels, Belgium
ghulin@prlb2.uucp

Abstract

This paper presents a parallel algorithm for recursive query processing and shows how it can be efficiently implemented in a local computer network. The algorithm relies on an interpretive approach where recursive rule processing and data retrieval are merged in a top-down computation. It employs "sideways information passing" to restrict to relevant facts the information extracted from the relational database. Evaluation is divided into a compilation phase and a dynamic phase. The compilation phase statically constructs a derivation tree that expresses the decomposition of a query into subqueries and the "sideways information passing" strategy. In the dynamic phase, cooperative processes are associated with subsets of "equivalent" nodes of the derivation tree. They communicate by message passing without sharing memory. Some optimizations are discussed for a practical parallel implementation. Gains in efficiency with respect to classical sequential algorithms are also discussed.

1 Introduction

A *distributed database* [5] is a collection of data which are distributed over different computers of a computer network. Relations are partitioned into fragments vertically and/or horizontally. The *vertical fragmentation* of a relation is the subdivision of its attributes into groups; fragments are obtained by projecting the relation on each group. The *horizontal fragmentation* of a relation consists in partitioning its tuples into subsets.

Query evaluation in distributed databases is performed in parallel through the computer network. The existence of several cooperating processors results in increasing performance.

Databases can be given deductive capabilities by the addition of *deduction rules*. Rules are often restricted to be definite Horn clauses. They define new relations and can be recursive.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

This paper presents a parallel algorithm for evaluating atomic queries in deductive databases and shows how it can be efficiently implemented in a local computer network (locality is necessary because parallelism in the algorithm is relatively fine-grained). It suggests a multi-computer architecture well-fitted for distributed deductive databases where some processors handle fragments of the distributed database while others are devoted to the evaluation of recursive queries. Such an architecture is currently being developed in the database tract of the PRISMA project (PRISMA for PaRallel Inference and Storage MAchine) [13].

From a logical point of view, a *deductive database*[1] is a finite set of function-free definite clauses. It is composed of an *extensional* database and of an *intensional* database. The extensional database is a set of ground atomic formulas called *facts*. The intensional database is a set of *rules*.

Example 1

Consider a directed graph with two types of arcs and two predicates $e_1(X,Y)$ and $e_2(X,Y)$ meaning that there exists an arc, respectively of type 1 or 2, from Node X to Node Y. Then, the following rules

$p(X,Y) \leftarrow e_1(X,Y)$,

$p(X,Y) \leftarrow e_2(X,Z), p(Z,T), e_2(T,Y)$,

define a virtual predicate p true for couples of nodes (X,Y) such that there exists a path from X to Y all of whose arcs are of type 2 except the central one. •

It can be assumed, without loss of generality, that a fact in the extensional database and an atomic formula in the head of a rule of the intensional database are never instances of the same predicate. The predicates of the deductive database some of whose instances are heads of rules of the intensional database are called *virtual predicates*, the other ones are called *base predicates.*

The semantics of a deductive database B is given by its least Herbrand model $\mathcal{M}_B$ which can be constructed by fixpoint computation. A closed formula F is true in the deductive database B if $\models_{\mathcal{M}_B} F$.

A first approach for evaluating queries in deductive databases consists in adding to the traditional relational algebra a least fixpoint operator *LFP*. If p is a virtual predicate, $LFP(p) = \{a \mid \models_{\mathcal{M}_B} p(a)\}$.

In Example 1, the query $p(a,Y)$? is translated into

[1] For more details about deductive databases, see [12].

$\sigma_{1=a}LFP(p)$. Unfortunately, commuting selection and least fixpoint operator is not possible in general and that leads, for Example 1 and the query $p(a, Y)$?, to the necessity of computing the complete extension of p. Yet, in order to answer the query, only part of the extension of p is strictly necessary. The facts that effectively contribute to the answer are said to be *relevant* to the query. A good evaluation strategy should not extract other facts from p. In general, which facts are relevant also depends on an ordering of the predicates in the right-hand sides of the rules [4].

Various proposals have been made to overcome this problem [7,8,10,15,20,27] but they are restrictive in that they are complete only for special (typically linear) types of recursion.

Another approach takes advantage of the decomposition of queries into subqueries and of the possible constant propagation among them. This approach seems more fruitful and has been heavily investigated recently. In [16], an active connection graph is dynamically created, with rule nodes and goal nodes corresponding to queries and subqueries. Solutions flow across the rule nodes to the goal nodes. However, constant propagation is poor and therefore, the algorithm is rather inefficient. Also the APEX algorithm [14] clearly lacks an efficient propagation of constants. The Recursive Query/Subquery (QSQR) and QoSaQ approaches [24,25], the Alexander method [19], and the generalized counting method [4] are algorithms which compute only relevant facts.

A uniform presentation of recursive query evaluation methods can be found in [18]. The algorithms are classified in three classes according to their halting condition. They are also compared on completeness and efficiency criteria.

Another efficiency requirement, which has received little attention until now, is the minimization of the number of database accesses. It can be addressed in two ways. First, when the same query on base predicates is generated at different times, a good evaluation process should access the database only once. Further, the strategies mentioned above repeatedly instantiate subsets $\{x_{i_1}, ..., x_{i_k}\}$ $(k \leq n)$ of variables in base predicate $p(x_1, ..., x_n)$, and access the extensional database for all the facts $p(u_1, ..., u_n)$ such that $(u_{i_1}, ..., u_{i_k})$ are the values instantiating $(x_{i1}, ..., x_{ik})$. It is clear that, rather than accessing the extensional database each time an instance is generated, a better strategy should accumulate successive instances and make a single access for solving all of them simultaneously.

The algorithms cited above are sequential and little attention has been paid to parallelism until now. Parallelism is however recognized as a very important optimization feature for recursive query evaluation. A few proposals exist for evaluating transitive closures in distributed database systems [1,9,22]. More generally, several classes of logic programs, e.g., the class of linear single rule programs, can be evaluated in parallel without introducing interprocess communication, or synchronization overhead [26]. A first step towards a parallel algorithm for general recursive queries has been proposed in [23]. A first attempt to implement that algorithm on top of a multi-computer system [21] has not proved satisfactory, due to an excess of duplicated work.

This paper is a continuation of [11] where we presented a new sequential algorithm, in the same vein as the QSQR approach, that accepts general recursion and guarantees termination and completeness. This paper describes a parallel version of the algorithm. The evaluation of a query is decomposed into two parts. The first one is static: it is a compilation of the query into an AND/OR tree. The second one is dynamic: cooperative processes associated with nodes of the tree rule the flow of data traversing the tree. They communicate by message passing and do not share memory. We also discuss possible optimizations for a practical parallel implementation.

The paper is organized as follows. Definitions and notations are introduced in the following section. The compilation phase is described in Section 3 while Section 4 presents the parallel processes of the dynamic phase. Section 5 discusses some possible optimizations: choice of a selection function, minimisation of the number of database accesses, and increase of parallelism. Section 6 is a conclusion.

2 Basic definitions

Variables in rules are taken in an alphabet V not containing the symbol *. In addition, V^* is a new set of variables where x^* is an element of V^* if and only if x is an element of V.

A *query scheme* associated with a predicate p is any atomic formula $p(x_1, ..., x_n)$ without constants and whose variables are in V^* or in V. Two variables $x \in V$ and $x^* \in V^*$ cannot both be present in the same query scheme. If x^* is present, x is called an *entry variable* of the query scheme. The *exit variables* of a query scheme are those belonging to V.

Two query schemes associated with an n-ary predicate p are said to be *equivalent* if they are identical up to a renaming of their variables respecting their entry or exit character.

An *entry* (resp., *exit*) *value* for a query scheme with n entry (resp., exit) variables is any set of n variable/value pairs where each variable appears once and only once.

A mechanism is assumed, provided for example by a (distributed) DBMS, that, given a query associated with an n-ary base predicate, computes its solution from the extensional database. More precisely, if Q is a query scheme associated with a base predicate, *in-v* is an entry value of Q, and B_E is the database, then,

$$ans(Q, in\text{-}v, B_E) = \{in\text{-}v \cup out\text{-}v \mid out\text{-}v \text{ is an exit value of } Q \ \& \ Q[in\text{-}v \cup out\text{-}v] \in B_E\}$$

where $F[\sigma]$ denotes the result of applying the substitution σ to the formula F.

3 The compilation phase

The result of compiling a query scheme is a *derivation tree* whose nodes are labeled with query schemes.

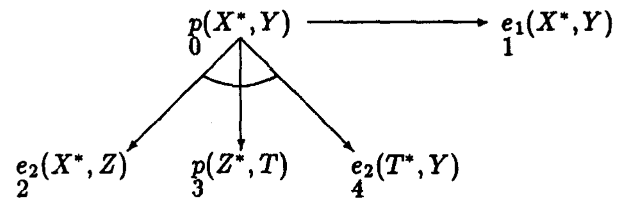
Derivation trees are special kinds of AND/OR trees [17]. They can be viewed as a kind of hypertree, where, instead of arcs connecting pairs of nodes, hyperarcs connect a parent node to an ordered list of successor nodes. The derivation tree of a query scheme makes explicit its possible decompositions (derived from the deduction rules) into subquery schemes. Like in Prolog, the ordering of subqueries governs the propagation of the instantiations of the entry variables.

The compilation phase constructs the derivation tree by recursively splitting each query scheme in as many sequences of subquery schemes as there are deduction rules defining the query scheme predicate in the intensional database. The resulting derivation tree has the following properties:

- when several equivalent query schemes 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. Different selection functions can generate different derivation trees. A good choice of the function is important: the efficiency of the evaluation process is very sensitive to the ordering. Heuristics guiding the choice of a selection function are discussed in Section 5.1;
- a variable in a subquery scheme SQ at the extremity of a hyperarc h is an entry variable if it is an entry variable of the query scheme which is the origin of h or an exit variable of a subquery scheme preceding SQ in the extremity of h.

Example 2

For the query scheme $p(X^*, Y)$ and the database of Example 1, the following tree is a possible derivation tree (a hyperarc is denoted by several binary arcs joined together by a curved line and nodes are numbered for easier reference.):



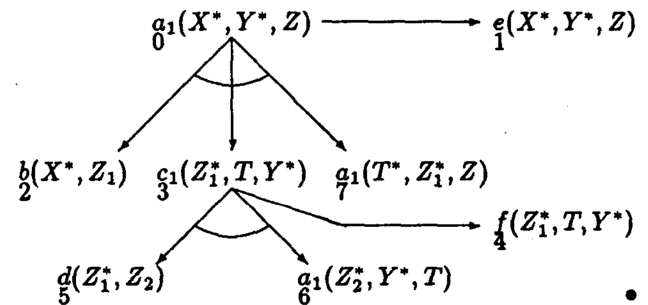
Note, for example, that Z which is an exit variable at Node 2 becomes an entry variable at Node 3. •

Example 3

In the following database, the virtual predicates a_1 and c_1 are mutually recursive:

$a_1(X, Y, Z) \leftarrow b(X, Z_1), a_1(T, Z_1, Z), c_1(Z_1, T, Y).$
$c_1(X, Y, Z) \leftarrow d(X, Z_1), a_1(Z_1, Z, Y).$
$a_1(X, Y, Z) \leftarrow e(X, Y, Z).$
$c_1(X, Y, Z) \leftarrow f(X, Y, Z).$

The predicates b, d, e and f are base predicates. The query scheme $a_1(X^*, Y^*, Z)$ leads to the possible derivation tree:



•

Before considering the dynamic phase, some more definitions are needed. A node n_1 of the derivation tree *precedes* a node n_2 (or, equivalently, n_2 *follows* n_1) if they are extremities of the same hyperarc and if n_1 is on the left of n_2. In Example 3, Node 2 precedes Node 3 and Node 7; Node 5 precedes Node 6. A node without a predecessor is called an *initial* node (Nodes 0, 1, 2, 4 and 5) and a node without a successor a *terminal* node (Nodes 0, 1, 4, 6 and 7).

The concept of *father node* has its usual meaning: Node 0 is the father of Nodes 1, 2, 3, and 7.

In the derivation tree, it may happen that several nodes are labeled with equivalent query schemes. The first one encountered during a depth-first traversal of the tree is called *archetype* node. In Example 3, Nodes 0, 1, 2, 3, 4 and 5 are archetype nodes. This concept will be useful for the dynamic phase.

4 The dynamic phase

An *atomic query* is a query scheme together with an entry value for it. It is denoted by the query scheme where entry variables have been instantiated with the

entry value. The dynamic phase operates on an atomic query and the derivation tree of its scheme, as constructed during the compilation phase, and produces a set of exit values for the query scheme. This set is computed step-by-step by repeatingly traversing the tree.

The dynamic phase consists of the interleaving of parallel processes. Each process executes in a dynamic *context*. A context remembers the minimal part of the computation state necessary to carry on with the computation. *Active queries* remember the subqueries generated during the evaluation process with their partial solution and the contexts in which they were issued. Contexts and active queries play an essential role in avoiding redundant work as well as in guaranteeing correctness and completeness. They are presented in Section 4.1. Parallel processes are informally described in Section 4.2 and their precise algorithms are given in Section 4.3.

4.1 Contexts and active queries

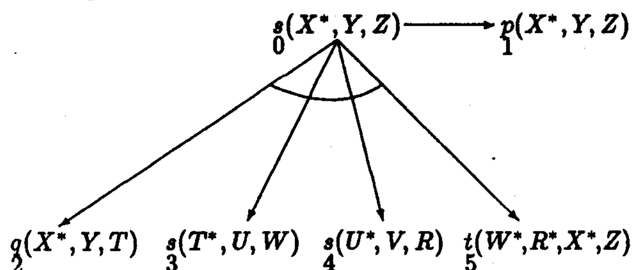
A context of invocation of a process at a node n has the form <n,*rel-v*,*in-v-f*> where

- *in-v-f* is an entry value for n's father node ({} if n is the root node),
- *rel-v* is a *relevant value* of n, i.e., a set of variable/value pairs which comprises the values of
  - entry variables of nodes which follow n in the extremity of a hyperarc provided those variables are also entry variables of the father of n or exit variables of nodes preceding n,
  - exit variables of the father of n which are exit variables of nodes preceding n.

The variables in a relevant value of a node n are called *relevant variables* of n. During evaluation, the relevant value in a context of n remembers the values of variables that are known before evaluating the current query at n and that are still needed after its evaluation.

To illustrate these concepts, consider the following derivation tree:

Example 4



At Nodes 0 and 1, the set of relevant variables is empty, at Node 2, it is $\{X\}$, at Node 3, it is $\{X, Y\}$, at Node 4, $\{X, Y, W\}$ and at Node 5, $\{Y\}$.

At Node 4, a context comprises, besides a reference to Node n itself:

- a relevant value $\{Y/c_1, X/c_2, W/c_3\}$. Y/c_1 is a part of a possible solution of the query at Node 0 with entry value $\{X/c_2\}$ while X/c_2 and W/c_3 are needed at Node 5 to find solutions for Z,
- an entry value for Node 0: $\{X/c_2\}$.

From this context and an exit value from Node 4, a context and an entry value can be constructed for Node 5. A context and an exit value of Node 5 allow to construct an exit value of Node 0. A context of a node remembers the minimal part of the computation state necessary to go on in the computation when an exit value is found at the node. •

During the evaluation process, information units, called *active queries*, 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. As distinct nodes may be labeled with equivalent query schemes, it is enough to associate active queries only with archetype nodes.

An active query at an archetype node n comprises:

- an entry value (it uniquely identifies the active query at the node),
- a set of exit values for that entry value (this set will be used to avoid repeating the same computation and, in particular, to avoid cycling),
- a set of contexts of nodes whose archetype is n (this set will be used to guarantee completeness, that is, to guarantee that exit values are used in all the contexts in which they were requested).

During the evaluation process, the set of exit values of the active queries are not necessarily complete, i.e. do not contain all the solutions of the query. That is why the contexts where queries are generated are remembered: each time a new solution is found for an active query of a node n_i, it is propagated along all contexts remembered in that active query. This guarantees completeness: each exit value of each active query is propagated along each context. At the end of evaluation, each active query has received a complete set of exit values.

Query schemes at equivalent nodes differ by their variables. If n and n' are equivalent nodes and v is an entry or exit value for n, the corresponding value for n' will be denoted $v@n'$. In Example 3, Nodes 0 and 7 are equivalent. An entry value of Node 7 is $<T/a, Z_1/b>$. Then, $<T/a, Z_1/b>@0 = <X/a, Y/b>$ is an entry value of Node 0.

4.2 Parallel processes

There is one process **Evaluate**(n) for each archetype node n. They interactively solve subqueries generated during evaluation.

To each archetype node n_a we associate:

- a process **Evaluate**(n_a) which computes and propagates the answers of active queries at n_a,
- a private memory *Memory*(n_a), that is, a set of active queries of n_a,
- two buffers, if n_a is a virtual node, to store messages sent by other processes:
  Request(n_a), a set of *request messages* for n_a,
  Answer(n_a), a set of *answer messages* for n_a,
- a buffer *Request*(n_a), if n_a is a base node.

A *request message* for n_a is a pair made of an entry value and a context for a node whose archetype is n_a. An *answer message* is a set of variable/value pairs for the variables of n_a.

In Example 4, four processes coexist: **Evaluate**(n_i) for $i = 0, 1, 2, 5$. A query such as $s(x, Y, Z)$ (lower case letters denote constants) is solved by executing:

$$\mathbf{Evaluate}(n_0) \parallel \ldots \parallel \mathbf{Evaluate}(n_5)$$

after having sent the message $<\{X/x\},<n_0,\{\},\{\}>>$ to *Request*(n_0). $P_1 \parallel P_2$ denotes the parallel computation of P_1 and P_2.

The reception of $<\{X/x\},<n_0,\{\},\{\}>>$ in buffer *Request*(n_0) initiates the computation. The process **Evaluate**(n_0) adds to *Memory*(n_0) the active query $(\{X/x\},\{\},\{<n_0,\{\},\{\}>\})$. That query is solved by parallel decomposition along each hyperarc issuing from node 0 as follows:

1. A request message $<\{X/x\},< n_1,\{\},\{X/x\}>>$ is sent to *Request*(n_1). Node n_1 is a base node. The request is solved by **Evaluate**(n_1) which computes $S = ans(p(X^*, Y, Z), \{X/x\}, B_E)$ and sends it to *Answer*(n_0). That computation is remembered by adding to *Memory*(n_1) the active query $(\{X/x\}, S, \{<n_1,\{\},\{X/x\}>\})$. If, later, n_1 is queried again with the same entry value X/x, the set of answers is already present in *Memory*(n_1) and no new access to the extensional database is necessary.
2. A request message $<\{X/x\},<n_2,\{X/x\},\{X/x\}>>$ is sent to *Request*(n_2). Exit values $\{Y/y, T/t\}$ are computed by **Evaluate**(n_2) by accessing B_E for q (see 1.). For each of them, the request message $<\{T/t\},<n_3,\{X/x,Y/y\},\{X/x\}>>$ is sent to *Request*(n_0), since n_0 is the archetype of n_3.
   Two situations may then arise, in general, for **Evaluate**(n_0):
   - there exists in *Memory*(n_0) an active query Q with entry value $\{X/t\}$. In that case, if the context $<n_3,\{X/x,Y/y\},\{X/x\}>$ is not already in the set of contexts of Q, it is added to it. For each $\{Y/y_1, Z/z_1\}$ in the set of exit values of Q, the request message $<\{U/y_1\},<n_4,\{X/x,Y/y,W/z_1\},\{X/x\}>>$ is sent to *Request*(n_0)(n_0 is the archetype of n_4),
   - otherwise, an active query with the entry value $\{X/t\}$, an empty set of exit values and a singleton set of contexts containing $<n_3,\{X/x,Y/y\},\{X/x\}>$ is added to *Memory*(n_0). Its set of exit values is computed by parallel decomposition as described above for the original query $s(x, Y, Z)$.
3. Each $<\{U/y_1\},<n_4,\{X/x,Y/y,W/z_1\},\{X/x\}>>$ in *Request*(n_0) is also solved by parallel decomposition. Each answer $\{U/y_1, V/v, R/r\}$ generates a request $<\{W/z_1,R/r,X/x\},<n_5,\{Y/y\},\{X/x\}>>$ in *Request*(n_5). These requests are solved by **Evaluate**(n_5) by accessing B_E for $t(z_1, r, x, Z)$. Each value z_2 of Z resulting from that evaluation generates an answer message $\{X/x, Y/y, Z/z_2\}$ sent to *Answer*(n_0).

Answer messages are processed in parallel with request messages. Notice that two active queries in the private memory of a process have different entry values. If a query is encountered more than once at equivalent nodes, there will be, in the private memory of the process associated with their archetype node, an active query with several contexts to remember the different states where the queries appeared. The contexts of a query $(\{X/a\}, S, C)$ at n_0 are of one of the form:

a. $<n_0,\{\},\{\}>$,
b. $<n_3,\{X/x,Y/y\},\{X/x\}>$,
c. $<n_4,\{X/x,Y/y,W/w\},\{X/x\}>$.

During the evaluation process, the set of exit values of the active queries are not necessarily complete, i.e. do not contain all the solutions of the query (except for queries on base predicates). That is why the contexts where queries are generated are remembered: each time a new solution is found for an active query of *Memory*(n_i), **Evaluate**(n_i) propagates it along all contexts remembered in that active query. This guarantees completeness: each exit value of each active query is propagated along each context. At the end of evaluation, each active query has received a complete set of exit values.

Given a solution $\{X/a, Y/b, Z/c\}$ of *Answer*(n_0) and the active query $(\{X/a\}, S, C)$ at Node n_0, if $\{Y/b, Z/c\}$ is not already in the current solution S, it is added to S and the solution is propagated in each context in C of the form b or c[2], i.e.,

type b: $<\{U/b\},<n_4,\{X/x,Y/y,W/c\},\{X/x\}>>$ is sent to *Request*(n_0),
type c: $<\{W/w,R/c,X/x\},<n_5,\{Y/y\},\{X/x\}>>$ is sent to *Request*(n_5).

[2]There is no propagation in the context of type a which is the context of the initial query.

Termination occurs when all request and answer buffers are empty and when all processes are waiting for new messages.

4.3 Algorithms

Some properties of entry, exit and relevant variables will be useful for writing the algorithms:

1. The entry and relevant variables of an initial node are also entry variables of its father.
2. The entry and relevant variables of a node with a predecessor are among the exit and relevant variables of the predecessor.
3. If n is terminal, then n's relevant and output variables make up the exit variables of n's father.

Let us call $\Theta^{ent}_n(vvp)$ those elements in a list of variable/value pairs *vvp* whose variables are entry variables of n and $\Theta^{rel}_n(vvp)$ those elements whose variables are relevant to n. With these notations, the preceding properties write:

1. if n is an initial node and *in-v-f* is an entry value of n's father, $\Theta^{ent}_n(in\text{-}v\text{-}f)$ is an entry value of n and $\Theta^{rel}_n(in\text{-}v\text{-}f)$ is a relevant value of n,
2. if n has a successor *n-suc*, if *rel-v* is a relevant value of n and *out-v* is an exit value of n, $\Theta^{ent}_{n\text{-}suc}(rel\text{-}v \cup out\text{-}v)$ is an entry value of *n-suc* and $\Theta^{rel}_{n\text{-}suc}(rel\text{-}v \cup out\text{-}v)$ is a relevant value of *n-suc*,
3. if n is terminal and if *rel-v* is a relevant value of n and *out-v* is an exit value of n, $rel\text{-}v \cup out\text{-}v$ is an exit value of n's father.

All the elements necessary to present the processes are now gathered[3].

If n_a is a virtual archetype node:

Evaluate(n_a):

While *true* do

1. if $Request(n_a) \neq \{\}$, then call **action**$_1(n_a)$; *
2. if $Answer(n_a) \neq \{\}$, then call **action**$_2(n_a)$. *

action$_1(n_a)$:

1. take an element $<in\text{-}v, c>$ ($c = <n, rel\text{-}v, in\text{-}v\text{-}f>$) out of $Request(n_a)$;
2. if there exists an active query $Q = (in\text{-}v@n_a, S, C)$ in $Memory(n_a)$,
   then,
   if c is not in C,
   1. add c to C;
   2. for each *out-v* in S,
      call **propagate**($out\text{-}v@n, c$)

   else,
   1. create the new active query
      $NQ = (in\text{-}v@n_a, \{\}, \{c\})$;
   2. add NQ to $Memory(n_a)$;
   3. for each hyperarc h issuing from Node n_a, *
      let *n-in* be the origin of h, *
      $n\text{-}in_a$ be the archetype of *n-in*, *
      send $<\Theta^{ent}_{n\text{-}in}(in\text{-}v@n_a), <n\text{-}in, \Theta^{rel}_{n\text{-}in}(in\text{-}v@n_a), in\text{-}v@n_a>>$ to $Request(n\text{-}in_a)$. *

action$_2(n_a)$:

1. take an element s out of $Answer(n_a)$:
2. let $Q = (in\text{-}v, S, C)$ be the active query of $Memory(n_a)$ such that $in\text{-}v \subset s$
   if $out\text{-}v = s \backslash in\text{-}v$ is not in S,
   then,
   1. add *out-v* to S;
   2. for each $c = <n, rel\text{-}v, in\text{-}v\text{-}f>$ in C
      call **propagate**($out\text{-}v@n, c$).

propagate($out\text{-}v, <n, rel\text{-}v, in\text{-}v\text{-}f>$):

If n is a terminal node,
then,
 if n has a father $n\text{-}f_a$,
 then
  send $in\text{-}v\text{-}f \cup rel\text{-}v \cup out\text{-}v$ to $Answer(n\text{-}f_a)$,
else,
 let *n-suc* be the successor of n,
  $n\text{-}suc_a$ be the archetype of *n-suc*,
 send $<\Theta^{ent}_{n\text{-}suc}(rel\text{-}v \cup out\text{-}v), <n\text{-}suc, \Theta^{rel}_{n\text{-}suc}(rel\text{-}v \cup out\text{-}v), in\text{-}v\text{-}f>>$ to $Request(n\text{-}suc_a)$.

The process associated with a base archetype Node n_a is similar to the one associated with a virtual node except that the lines of **Evaluate**(n_a) and **action**$_2(n_a)$ marked by *'s are respectively replaced by:

if $Request(n_a) \neq \{\}$, then call **action**$_1(n_a)$.

and

3. let Q be the query scheme labeling n_a,
   1. replace the empty set of exit values of NQ by $ans(Q, in\text{-}v@n_a, B_E)$;
   2. for each s in $ans(Q, in\text{-}v@n_a, B_E)$,
      call **propagate**($(s@n) \backslash in\text{-}v, c$).

As they are presented, the algorithms implement a tuple-oriented evaluation strategy: a message only contains one answer or one request. This is not inherent to our evaluation method. Each process could also accumulate the requests and answers it generates until its own *Answer* and *Request* buffers are empty and then only send sets of requests and answers to the other processes. The evaluation would then be set-oriented.

Any other strategy that lies between those two extremes is also possible. The choice of an optimal strategy depends on the communication costs in the distributed architecture. If they are negligible, a tuple-oriented strategy can be preferable because, as each request and answer are sent as soon as they are generated, the cpu utilization of the processes is optimized.

[3] Correctness and completeness of the corresponding sequential algorithm have been proved in [11]. The demonstration could easily be transformed for the parallel version.

Otherwise, a more set-oriented strategy must be chosen.

4.4 Termination

In order to discuss termination, each process is assumed to acknowledge each message it receives as soon as it reads it and to maintain, in a counter, the number of messages that it sent and has not yet been acknowledged of.

Termination occurs as soon as, for every process,

- *Request* and *Answer* buffers are empty,
- there is no message currently being processed,
- the value of counter is equal to zero.

The problem in a distributed architecture is determining asynchronously when all the above conditions are satisfied. Fortunately, that problem has well-known solutions. Their presentations is out of the scope of this paper and the interested reader is invited to refer to [6]. That problem is also shortly discussed in [23].

4.5 Example

We now present the trace of an example. It is short but typical due to the symmetry of the e_2 relation on the couple (a, b). This symmetry is responsible for the formation of an active query with two different contexts. We have made explicit the consequences of the execution of processes. The creations and alterations of the different active queries at the root node are indicated in $M(n_0)$ (for *Memory*(n_0)). For the sake of simplicity and brevity, they are not shown for the base archetype nodes. The *Request* and *Answer* buffers of a node n are denoted $R(n)$ and $A(n)$. They are empty by default. In this simple example, entry and exit values have only one component.

The intensional database is

$p(X,Y) \leftarrow e_1(X,Y)$

$p(X,Y) \leftarrow e_2(X,Z), p(Z,T), e_2(T,Y).$

The extensional database is

$e_1(b,c)$	$e_2(a,b)$	$e_2(d,e)$
$e_1(d,g)$	$e_2(b,a)$	$e_2(e,f)$
	$e_2(c,d)$	$e_2(g,h)$

The query is *p(a,Y)*. Nodes n_i $(0 \leq i \leq 4)$ are numbered as in Example 2. Nodes n_0, n_1 and n_2 are archetype nodes.

Initially, $R(n_0) = \{<\{X/a\},<n_0,\{\},\{\}>>\}$.

1. **Evaluate**(n_0)
$M(n_0) = \{(\{X/a\},\{\},\{<n_0,\{\},\{\}>\})\}$
$R(n_1) = \{<\{X/a\},<n_1,\{\},\{X/a\}>>\}$
$R(n_2) = \{<\{X/a\},<n_2,\{\},\{X/a\}>>\}$

2. **Evaluate**(n_1) || **Evaluate**(n_2)
$R(n_0) = \{<\{Z/b\},<n_3,\{\},\{X/a\}>>\}$

3. **Evaluate**(n_0)
$M(n_0) = \{(\{X/a\},\{\},\{<n_0,\{\},\{\}>\}),$
$(\{X/b\},\{\},\{<n_3,\{\},\{X/a\}>\})\}$
$R(n_1) = \{<\{X/b\},<n_1,\{\},\{X/b\}>>\}$
$R(n_2) = \{<\{X/b\},<n_2,\{\},\{X/b\}>>\}$

4. **Evaluate**(n_1) || **Evaluate**(n_2)
$R(n_0) = \{<\{Z/a\},<n_3,\{\},\{X/b\}>>\}$
$A(n_0) = \{\{X/b,Y/c\}\}$

5. **Evaluate**(n_0)
$M(n_0) = \{(\{X/a\},\{\},\{<n_0,\{\},\{\}>,<n_3,\{\},\{X/b\}>\}),$
$(\{X/b\},\{\{Y/c\}\},\{<n_3,\{\},\{X/a\}>\})\}$
$R(n_2) = \{<\{T/c\},<n_4,\{\},\{X/a\}>>\}$

6. **Evaluate**(n_2)
$A(n_0) = \{\{X/a,Y/d\}\}$

7. **Evaluate**(n_0)
$M(n_0) =$
$\{(\{X/a\},\{\{Y/d\}\},\{<n_0,\{\},\{\}>,<n_3,\{\},\{X/b\}>\}),$
$(\{X/b\},\{\{Y/c\}\},\{<n_3,\{\},\{X/a\}>\})\}$
$R(n_2) = \{<\{T/d\},<n_4,\{\},\{X/b\}>>\}$

8. **Evaluate**(n_2)
$A(n_0) = \{\{X/b,Y/e\}\}$

9. **Evaluate**(n_0)
$M(n_0) =$
$\{(\{X/a\},\{\{Y/d\}\},\{<n_0,\{\},\{\}>,<n_3,\{\},\{X/b\}>\}),$
$(\{X/b\},\{\{Y/c\},\{Y/e\}\},\{<n_3,\{\},\{X/a\}>\})\}$
$R(n_2) = \{<\{T/e\},<n_4,\{\},\{X/a\}>>\}$

10. **Evaluate**(n_2)
$A(n_0) = \{\{X/a,Y/f\}\}$

11. **Evaluate**(n_0)
$M(n_0) =$
$\{(\{X/a\},\{\{Y/d\},\{Y/f\}\},\{<n_0,\{\},\{\}>,$
$<n_3,\{\},\{X/b\}>\}),$
$(\{X/b\},\{\{Y/c\},\{Y/e\}\},\{<n_3,\{\},\{X/a\}>\})\}$
$R(n_2) = \{<\{T/f\},<n_4,\{\},\{X/b\}>>\}$

12. **Evaluate**(n_2)

The computation stops here and the set of answers is $\{\{Y/d\},\{Y/f\}\}$.

5 Optimizations

5.1 The selection function

The selection function is of great importance because it orders the subquery schemes generated from a query scheme. The evaluation processes follow this order when they solve queries and propagate solutions.

In the rule $a(X,Y) \leftarrow b(X,Z), a(Z,Y)$, and for solving the query scheme $a(X^*,Y)$, a selection of $a(Z,Y)$ before $b(X^*,Z)$ leads to the computation of the complete relation a, which is clearly unnecessary. A good selection function must order the subquery schemes so that the number of relevant facts extracted from the extensional database becomes minimum.

A simple choice reducing the size of the manipulated relations consists of always selecting the literal with most entry variables, making the assumption (heuristic) that queries with more instantiated variables have

smaller set of exit values. Statistics could also be of great help.

Another selection practice is based on *recursion levels*. Base predicates are by convention at level 0. A predicate p is of higher level than a predicate q if an instance of q is in the antecedent of a rule defining p or if there exists an r such that p is of higher level than r and r than q. Of course, p and q are at the same level if p is of higher level than q and conversely.

A *recursion level* is a maximal set of predicates at the same level.

Along this line, a good selection practice is often to prefer subqueries leading to lowest recursion levels. Thus, we propose to select, in order of preference, the base predicates (level 0), then the predicates of strictly lower level than the consequent of the rule, and finally the ones at the same level. The assumption is that before working at a level of recursion, it is better to have computed a maximum of facts from lower levels so that maximum information is available.

A more sophisticated strategy integrates the two selection methods just outlined. It needs the definition of the concept of *basic determined part* of a set of query schemes [10]. If $\{l_1, l_2, \ldots, l_n\}$ is a set of query schemes, l_i $(0 \leq i \leq n)$ is *basic determined* if it is a base literal and if either it has an entry variable as one of its arguments or there exists a j $(0 \leq j \leq n)$ such that l_j is basic determined and such that l_i shares variables with l_j. The basic determined part of a set of query schemes is the set of basic determined query schemes.

The selection practice is then to take the entire basic determined part first. If it is empty, one among the virtual predicates of lowest level with a maximum number of entry variables is selected.

The basic determined part gathers together all the base literals which could have been chosen successively by the first simple selection methods. It is clearly more efficient to treat them as a single conjunctive query for accessing the extensional database instead of making successive elementary accesses. This can dramatically reduce the number of accesses. Moreover, conjunctive queries correspond to join operations, that a good DBMS can handle efficiently.

To accommodate this optimization, the compilation phase must be slightly modified. The nodes of the derivation tree are then either virtual query schemes or conjunctions of base ones (in this case, they are tip nodes).

The selection function is also discussed in [23].

5.2 Minimization of the number of extensional database accesses

The preceding processes access the extensional database whenever a new base query is generated. It is clear that, due to the time needed by the (possibly distributed) DBMS to answer literal queries, there is a risk that requests pile up in the buffers *Request*(n_i) associated with base nodes while other buffers become empty.

A first optimization can be to access the extensional database with queries composed of a disjunction of literal queries. In Example 2, *Request*(n_2) can always be written in the form

$$Request(n_2) = \bigcup_{i=1}^{n}\{<\{X/x_i\}, <n_2, \{\}, \{X/x_i\}>>\} \cup \bigcup_{j=1}^{m}\{<\{Z/z_j\}, <n_4, \{\}, \{X/x'_j\}>>\}.$$

Instead of answering the requests separately, the DBMS can be accessed with the composed query $\bigvee_{i=1}^{n} e_2(x_i, Y) \vee \bigvee_{j=1}^{m} e_2(z_j, Y)$. The answers to the query are then sorted by **Evaluate**(n_2), added to the set of exit values of their corresponding active queries and propagated to *Request*(n_0) or returned to *Answer*(n_0) while another composed query is being solved by the DBMS. Thus, the number of accesses to the DBMS decreases and their efficiency increases.

Further, a special interface process can be in charge of managing all queries coming from processes associated with the base nodes. It could for example give a higher priority to the composed queries which are estimated to have the largest set of answers. The intuition underlying this strategy is that it is wise to extract the relevant information from the extensional database as early as possible in order to make it available for computing the virtual queries. Heuristics must be designed to implement that strategy. They could be based on statistical informations or more simply on the rough assumption that the larger a disjunctive composed query, the larger its set of answers. We will not go further in that kind of optimization, because it is clearly closely dependent on the architecture of the DBMS.

5.3 Increasing parallelism

Parallelism can be increased by associating cooperative processes with *Request* and *Answer* buffers instead of archetype nodes. Indeed, the **Evaluate**(n_a) process associated with a virtual archetype node n_a alternatively answers messages from *Request*(n_a) and *Answer*(n_a). It can clearly be split into two parallel processes separately managing the two buffers. Those two processes would share *Memory*(n_a) and some attention must be paid to their correct design.

If N_b and N_v are the numbers of archetypes nodes labeled with query schemes and respectively associated

with base and virtual predicates, then, the number of involved processes becomes $N_b + (2 * N_v)$ instead of $N_b + N_v$.

Parallelism can also be increased by a better use of the or-parallelism of the rules. In Example 2,

- requests in buffer *Request*(n_0) generate, through **Evaluate**(n_0), requests in buffers *Request*(n_1) and *Request*(n_2),
- requests in buffer *Request*(n_1) generate, through **Evaluate**(n_1), answers in buffer *Answer*(n_0),
- requests in buffer *Request*(n_2) generate, through **Evaluate**(n_2), answers in buffer *Answer*(n_0) and requests in buffer *Request*(n_0),
- answers in buffer *Answer*(n_0) generate, through **Evaluate**(n_0), requests in buffer *Request*(n_2).

Cooperative processes could be associated with those communication channels between buffers rather than with archetype nodes or *Request* and *Answer* buffers of archetype nodes. This would again result in an increase of the number of involved processes.

In Example 2, there are 3 archetype nodes, 4 *Request* and *Answer* buffers, and 6 communication channels between buffers.

6 Conclusion

Performance analysis for sequential recursive query evaluation strategies is not easy as shown in [3] where it has been carried out for several strategies on a set of four queries, over a range of data.

The analysis of parallel evaluation strategies is still more complex. For computing the answer to a query, one must take into account the number of messages, the size of buffers and private memories, the number of concurrent accesses to the extensional (distributed) database and so on. In full generality, this is not straightforward, if not impossible. So we will content ourselves with general considerations.

Our algorithm, in its sequential version [11], is close to the QSQR algorithm of [24]. In its spirit, it is also similar to the Alexander method [19] or to the similar Magic Sets method [2]. In the latter methods, constant propagation is compiled into new Horn rules instead of being interpreted. These algorithms are among the most efficient for general recursive rules and general data [3].

Our main contribution, besides the introduction of parallelism and the attention devoted to minimizing the number of database accesses, is the new concept of contexts which remember the minimal part of the computation state necessary to go on when new exit values are found at nodes of the derivation tree. In QSQR, evaluation of queries and subqueries is repeated from the beginning as long as there are no more new solutions. This leads to duplicated work which is avoided with our contexts. In [2] and [19], contexts are hidden in new predicates of rewritten rules. Albeit very elegant, such a solution increases the number of virtual predicates.

Further introducing parallelism is an undisputable speed-up. Of course, one cannot expect to gain an order of magnitude in the evaluation performance: the chosen architecture is multi-computer, with a finite and rather small number of processors. However, efficiency is significantly enhanced for the following reasons:

- Our algorithm fits quite naturally in the framework of a distributed database. It can take advantage of this framework where access to the extensional database is speeded up.
- Evaluation time heavily depends on the average number of processors in operation at any given moment. The higher is the branching in base relations, the higher the average number of working processors. In the worst case, this number is close to one. This is true for example when computing the ancestors of an individual from a parent relation where every individual has at most one child. Then, the problem is intrinsically sequential and no gain is obtained. In the most favorable case of relations with high branching, most processors are working together and parallel computation reaches its full power with a true improvement in execution time.
- In a sequential architecture, the time spent in accessing the extensional database is crucial. During these often numerous accesses, the sequential process suspends itself in a waiting state. It is a bottleneck that can be responsible for a very bad overall performance, as the total evaluation time is always much larger than the total access time to the database. In a parallel architecture, processors work simultaneously with the accesses to the database, evaluation time often remains close to the total access time. Furthermore, in Section 5.2, we showed that the total access time could be significantly decreased by gathering related queries as single access.

Our work can hardly be compared with [1,9,22]. Those papers are dedicated to the parallel evaluation of transitive closure of binary relations and cannot be generalized to general datalog programs. Moreover, [1] and [22] only can compute the complete transitive closure of a relation and thus do not focus on relevant data.

The approach of [26] is very attractive because it introduces parallelism without the necessity of interprocess communication. However, it is only applicable to a limited class of datalog programs (a superclass of the linear single rule programs).

Our work is closely related to [23] where evaluation

is also divided into a compilation phase that builds a rule/goal graph, and a top-down dynamic phase. We have simplified the compilation phase: our derivation graph is much more simple and less redundant than the rule/goal graph of [23]. The dynamic phase in [23] is very shortly and informally described. On the contrary, we tried to be as precise as possible and to provide clear guidelines and hints for implementation.

In conclusion, we have shown that a multi-computer architecture is a nice framework on top of which to implement distributed databases with deductive capabilities. Recursive evaluation with the help of our algorithm improves the traditional sequential methods.

ACKNOWLEDGEMENTS

This work was supported by the Commission of the European Communities, under Project ESTEAM-316 of the ESPRIT Program. We are grateful to A. Pirotte and D. Roelants, our colleagues of PRLB, for their careful readings of early drafts of this paper.

References

[1] R. Agrawal and H. V. Jagadish. Multiprocessor transitive closure algorithms. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Austin, Texas, Dec. 1988.

[2] F. Bancilhon, D. Mayer, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM Symposium on Principles of Database Systems*, March 1986.

[3] F. Bancilhon and R. Ramakrishnan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 439–518, Morgan Kaufmann, 1988.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of Sixth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[5] S. Ceri and G. Pelagatti. *Distributed Databases. Computer Science Series*, McGraw-Hill, 1984.

[6] M. Chandy and J. Misra. An example of stepwise refinement of distributed programs: quiescence detection. *ACM Transactions on Programming Language and Systems*, 8(3):326–343, July 1986.

[7] C. L. Chang. On the evaluation of queries containing derived relations in a relational data base. In H. Gallaire and J. Minker, editors, *Advances in Data Base Theory, vol. 1*, pages 235–260, Plenum Press, New York, 1980.

[8] G. Gardarin and C. de Maindreville. Evaluation of database recursive logic programs as recurrent function series. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 177–186, Washington, May 1986.

[9] K. Guh and C. T. Yu. Evaluation of transitive closure in distributed database systems. *IEEE Journal on Selected Areas in Communications*, 7(3):399–407, Apr. 1989.

[10] L. Henschen and S. Naqvi. On compiling queries in recursive first order databases. *Journal of the ACM*, 31(1):137–147, 1984.

[11] G. Hulin. *An Efficient Interpretive Algorithm for Recursive Queries*. Technical Report R521, Philips Research Laboratory Brussels, Jan. 1988.

[12] G. Hulin, A. Pirotte, D. Roelants, and M. Vauclair. Logic and databases. In A. Thayse, editor, *From Modal Logic to Deductive Databases*, Wiley, 1989.

[13] M. L. Kersten, P. M. Apers, M. A. W. Houtsma, E. J. A. van Kuijk, and R. L. W. van de Weg. A distributed, main-memory database machine. In *Proceedings of the Fifth International Workshop on Database Machines*, pages 496–512, Karuizawa, 1987.

[14] E. Lozinskii. Evaluating queries in deductive databases by generating. In *IJCAI'85 Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, USA*, pages 173–177, Aug. 1985.

[15] G. Marque-Pucheu, J. Martin-Gallausiaux, and G. Jomier. Interfacing prolog and relational data base management systems. In *ICOD-2 Workshop*, Cambridge, England, Sep. 1983.

[16] D. P. McKay and S. C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *IJCAI'81 Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver, B. C. Canada*, pages 368–375, 1981.

[17] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, 1980.

[18] D. Roelants. *Recursive Rules in Logic Databases*. Technical Report R513, Philips Research Laboratory Brussels, March 1987.

[19] J. Rohmer and R. Lescoeur. *La Méthode d'Alexandre : Une solution pour traiter les axiomes récursifs dans les bases de données déductives*. Research Report DRAL/IA/45.01, Centre de Recherche Bull, March 1985. (in French).

[20] D. Saccà and C. Zaniolo. The generalized counting method for recursive logic queries. In *Proceedings of the International Conference on Database Theory*, Roma, Sep. 1986.

[21] E. Smagge. *Distributed Evaluation in Pool-T of Horn Clauses*. PRISMA Doc. P0174, Philips Research Laboratories Eindhoven, Sep. 1987.

[22] P. Valduries. Parallel recursive query processing in a share nothing data server. In *Quatrième Journées Bases de Données Avancées*, pages 213–222, Benodet (France), May 1988.

[23] A. Van Gelder. A message passing framework for logical query evaluation. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 155–165, Washington, May 1986.

[24] L. Vieille. Recursive axioms in deductive databases: the Query/Subquery approach. In *Proceedings of the First International Conference on Expert Database Systems, Columbia, S.C.*, pages 253–267, Columbia, SC, 1986.

[25] L. Vieille. From QSQ towards QoSaQ: global optimization of recursive queries. In *Proceedings of the Second International Conference on Expert Database Systems, Washington, D.C.*, pages 421–436, Washington, DC, 1988.

[26] O. Wolfson. Sharing the load of logic programming evaluation. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 46–55, Austin, Texas, Dec. 1988.

[27] H. Yokota and S. K. et al. *An enhanced inference mechanism for generating relational algebra queries*. Technical Report TR-026, ICOT, Oct. 1983.