An Amateur's Introduction to Recursive Query Processing Strategies[†]

Francois Bancilhon

Raghu Ramakrishnan

Computer Sciences Technical Report #772

June 1988

[†]This paper is also to appear in books edited by M. Brodie, J. Mylopoulos, and by M. Stonebralur, to be published by Morgan-Kaufmann.

An Amateur's Introduction to Recursive Query Processing Strategies [†]

Francois Bancilhon (1)
Raghu Ramakrishnan (2)

(1): Altair
BP 105, 78153 Le Chesnay Cedex, France

(2): Computer Sciences Department
University of Wisconsin-Madison, Madison, 53706, USA

ABSTRACT

This paper surveys and compares various strategies for processing logic queries in relational databases. The survey and comparison is limited to the case of Horn Clauses with evaluable predicates but without function symbols. The paper is organized in three parts. In the first part, we introduce the main concepts and definitions. In the second, we describe the various strategies. For each strategy, we give its main characteristics, its application range and a detailed description. We also give an example of a query evaluation. The third part of the paper compares the strategies on performance grounds. We first present a set of sample rules and queries which are used for the performance comparisons, and then we characterize the data. Finally, we give an analytical solution for each query/rule system. Cost curves are plotted for specific configurations of the data.

1. Introduction

The database community has recently manifested a strong interest in the problem of evaluating "logic queries" against relational databases. This interest is motivated by two converging trends: (i) the desire to integrate database technology and artificial intelligence technology i.e., to extend database systems, to provide them with the functionality of expert systems thus creating "knowledge base systems" and (ii) the desire to integrate logic programming technology and database technology, i.e., to extend the power of the interface to the database system to that of a general purpose language. The second goal is of a somewhat different nature and has found in its ranks proponents of object oriented, functional and imperative as well as logic based programming languages. The logic programming camp is relying on the fact that logic programming and relational calculus have the same underlying mathematical model, namely first order logic.

Of course, database researchers already know how to evaluate logic queries: the view mechanism, as offered by most relational systems, is a form of support of a restricted set of logic queries. But those logic queries are restricted to be non-recursive and the problem of efficiently supporting recursive queries is still open.

In the past five years, following the pioneering work by Chang, Shapiro and McKay, and Henschen and Naqvi, numerous strategies have been proposed to deal with recursion in logic queries. The positive side of this work is that there are a lot of algorithms offered to solve *the* problem. The negative side is that we do not know how to make a choice of an algorithm. It seems reasonable to say that all these strategies can

[†] This is a revised version of a paper that appeared under the same title in the proceedings of SIGMOD 86.
The original paper was written while the first author was at MCC, Austin, and the second author was at the University of Texas at Austin and visiting MCC. This revision also includes an epilog covering work done since the paper was written.

only be compared on three grounds: functionality (i.e., application domain), performance and ease of implementation. However, each of these algorithms is described at a different level of detail, and it is sometimes difficult to understand their differences. In fact, we shall claim later in this paper that some of them are indeed identical. Each comes with little or no performance analysis, and the application domain is not always easy to identify. We try in this paper to evaluate these algorithms with respect to these three criteria. We describe all the algorithms at the same level of detail and demonstrate their behavior on common examples. This is not always easy to do since some of them are fairly well formalized while others are merely sketched as an idea.

For each one of them, we state in simple terms the application domain. Finally, we give a first simple comparison of the performance of these algorithms. Choosing a simple set of typical queries, a simple characterization of the data and a simple cost function, we give an analytical evaluation of the cost of each strategy. The results give a first insight into the respective value of all the proposed strategies.

The rest of the paper is organized as follows: In section 2 we present our definitions and notations, and introduce the main ideas. In section 3 we present the main features of the strategies, and describe each one individually, and finally, in section 4, we present the performance evaluation methodology and results.

2. Logic Databases

2.1. An Example

Let us start by discussing informally an example. Here is what we call a "logic database":

```
parent(cain,adam).
parent(abel,adam).
parent(cain,eve).
parent(abel,eve).
parent(sem,abel).
ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
generation(adam,1).
generation(X,I) :- generation(Y,J), parent(X,Y),J=I-1.
generation(X,I) :- generation(Y,J), parent(Y,X),J=I+1.
```

Note that this is a purely syntactic object. In this database, we have a set of predicate or relation names (parent, ancestor and generation), a set of arithmetic predicates (I=J+1, I=J-1) and a set of constants (adam, eve, cain, sem and abel). Finally, we have a set of variables (X,Y and Z). The database consists of a set of sentences ending with a period. "parent(cain,adam)" is a fact, and "ancestor(X,Y) :- parent(X,Y)" is a rule.

Let us now associate a meaning with the database. We first associate with each constant an object from the real world: thus, with "adam" we associate the individual whose name is "adam". Then, we associate with each arithmetic predicate name the corresponding arithmetic operator. Then we can interpret intuitively each fact and each rule. For instance we interpret "parent(cain,adam)" by saying that the predicate parent is true for the couple (cain,adam), and we interpret the rule

ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).

by saying that if there are three objects X, Y and Z such that ancestor(X,Z) is true and ancestor(Z,Y) is true then ancestor(X,Y) is true.

This leads to an interpretation which associates with each predicate a set of tuples. For instance with the predicate ancestor we associate the interpretation {(cain,adam), (abel,adam), (cain,eve), (abel,eve), (sem,abel), (sem,adam), (sem,eve)}, and with the predicate generation we associate the interpretation {(adam,1), (eve,1), (cain,2), (abel,2), (sem,3)}

The problem is to answer queries, given the logic database. For instance given a query of the form generation(sem,?) or ancestor(?,adam), how do we find the answer: generation(sem,3) and {ancestor(cain,adam), ancestor(abel,adam), ancestor(sem,adam)}?

Let us now formalize all the notions encountered in this example and define a logic database. We first define it syntactically, then we attach an interpretation to this syntax.

2.2. Syntax of a Logic Database

We first define four sets of names: *variable* names, *constant* names, *predicate* or *relation* names and *evaluable predicate* names.

We adopt the Prolog convention of denoting variables by strings of characters starting with an upper case letter and constants by strings of characters starting with a lower case letter or integers. For instance X1, Father and Y are variables, while john, salary and 345 are constants.

We use identifiers starting with lower case letters for predicates names and relation names (evaluable and non-evaluable).

We use the term relation (from database terminology) and predicate (from logic terminology) indifferently to represent the same object. We shall however interpret them differently: a relation will be interpreted by a set of tuples and a predicate by a true/false function. There is a fixed arity associated with each relation/predicate.

The set of evaluable predicate names is a subset of the set of predicate names. We will not be concerned with their syntactic recognition; in the examples it will be clear from the name we use. The main examples of evaluable predicate names are arithmetic predicates. For instance, sum, difference and greater-than are examples of evaluable predicates of arity 3, 3 and 2 respectively, while parent and ancestor are non-evaluable predicates of arity 2.

A *literal* is of the form p(t1,t2,...,tn) where p is a predicate name of arity n and each ti is a constant or a variable. For instance father(john,X), ancestor(Y,Z), id(john,25,austin) and sum(X,Y,Z) are literals. An *instantiated* literal is one which does not contain any variables. For instance id(john,doe,25,austin) is an instantiated literal, while father(john,Father) is not.

We allow ourselves to write evaluable literals using functions and equality for the purpose of clarity. For instance, Z = X+Y denotes sum(X,Y,Z), I = J+1 denotes sum(J,1,I), and X > 0 denotes greater-than(X,0).

If p(t1,t2,...,tn) is a literal, we call (t1,t2,...,tn) a *tuple*.

A *rule* is a statement of the form

```
        p :- q1,q2,...,qn.
```

where p and the qi's are literals such that the predicate name in p is a non-evaluable predicate. p is called the *head* of the rule, and each of the qi's is called a *goal*. The conjunction of the qi's is the *body* of the rule. We have adopted the Prolog notation of representing implication by ':-' and conjunction by ','. For instance

```
uncle(john,X) :- brother(X,Y), parent(john,Y).
```

is a rule with head "uncle(john,X)" and body "brother(X,Y), parent(john,Y)".

A *ground clause* is a rule in which the body is empty. A *fact* is a ground clause which contains no variables. For instance

```
        loves(X,john).
        loves(mary,susan).
```

are ground clauses, but only the second of these is a fact.

A *database* is a set of rules; note that this set is not ordered. Given a database, we can partition it into a set of facts and the set of all other rules. The set of facts is called the *extensional* database, and the set of all other rules is called the *intensional* database.

2.3. Interpretation of a Logic Database

Up to now our definitions have been purely syntactical. Let us now give an interpretation of a database. This will be done by associating with each relation name in the database a set of instantiated tuples. We first assume that with each evaluable predicate p is associated a set natural(p) of instantiated tuples which we call its *natural interpretation*. For instance, with the predicate *sum* is associated an infinite set of all the 3-tuples (x,y,z) of integers such that the sum of x and y is z. In general the natural interpretation of an evaluable predicate is infinite.

Given a database, an *interpretation* of this database is a mapping which associates with each relation name a set of instantiated tuples.

A *model* of a database is an interpretation I such that:

1. for each evaluable predicate p, I(p) = natural(p), and,
2. for any rule, p(t) :- q1(t1),q2(t2),...,qn(tn), for any instantiation σ of the variables of the rule such that σ(ti) is in the interpretation of qi for all i then σ(t) is is in the interpretation of p.

This is simply a way of saying that, in a model, if the right hand side is true then the left hand side is also true. This implies that for every fact p(x) of the database the tuple x belongs to the interpretation of p.

Of course, for a given database there are many models. The nice property of Horn Clauses is that among all these models there is a *least* one (least in the sense of set inclusion), which is the one we choose as *the* model of the database (Van Emden and Kowalski [1976]). Therefore from now on, when we talk about the model or the interpretation of a database, we mean its least model.

Notice that because of the presence of evaluable arithmetic predicates the minimal model is, in general, not finite.

Let p be an n-ary predicate. An *adornment* of p is a sequence *a* of length n of b's and f's (Ullman [1985]). For instance bbf is an adornment of a ternary predicate, and fbff is an adornment of predicate of arity 4. An adornment is to be interpreted intuitively as follows: the i-th variable of p is bound (respectively free) if the i-th element of *a* is b (respectively f). Let p(x1,x2,...,xn) be a literal, an adornment a1a2...an of that literal is an adornment of p such that:

(i) if xi is a constant then ai is b,
(ii) if xi = xj then ai = aj

We denote adornments by superscripts. A *query form* is an adorned predicate. Examples of query forms are father^{bf}, id^{bffb}.

A *query* is a query form and an instantiation of the bound variables. We denote it by an adorned literal where all the bound positions are filled with the corresponding constants and the free positions are filled by distinct free variables. Therefore father^{bf}(john,X) and id^{bffb}(john,X,Y,25) are queries. The distinction between queries and query forms are that query forms are actually compiled, and at run-time their parameters will be instantiated. Notice that father(X,X) is not a query form in this formalism.

The *answer* to a query q(t) is the set: {q(σ(t)) | σ is an instantiation of t, and σ(t) is in the interpretation of q}.

2.4. Structuring and Representing the Database

A predicate which only appears in the intensional database is a *derived* predicate. A predicate which appears only in the extensional database or in the body of a rule is a *base* predicate.

For performance reasons, it is good to decompose the database into a set of pure base predicates (which can then be stored using a standard DBMS) and a set of pure derived predicates. Fortunately, such a decomposition is always possible, because every database can be rewritten as an "equivalent" database containing only base and derived predicates. By equivalent, we mean that all the predicate names of the original database appear in the modified database and have the same interpretation.

We obtain this equivalent database in the following way: consider any predicate p that is neither base nor derived. By definition, we have a set of facts for p, and p appears on the left of some rules. So we simply introduce a new predicate p_ext and do the following:

1. replace p by p_ext in each fact of p,
2. add a new rule of the form p(X1,X2,...,Xn) :- p_ext(X1,X2,...,Xn) where n is the arity of p.

Example:

```
father(a,b).
parent(b,c).
grandfather(b,d).
grandfather(X,Y) :- father(X,Z),parent(Z,Y).
```

becomes:

```
father(a,b).
parent(b,c).
grandfather_ext(b,d).
grandfather(X,Y) :- father(X,Z),parent(Z,Y).
grandfather(X,Y) :- grandfather_ext(X,Y).
```

Most authors have chosen to describe a set of rules through some kind of graph formalism. Predicate Connection Graphs, as presented in McKay and Shapiro [1981], represent the relationship between rules and predicates. Rule/goal graphs, as presented in Ullman [1985], carry more information because predicates and rules are adorned by their variable bindings. We have chosen here to keep the rule/goal graph terminology while using unadorned predicates.

The *rule/goal* graph has two sets of nodes: square nodes which are associated with predicates, and oval nodes which are associated with rules. If there is a rule of the form

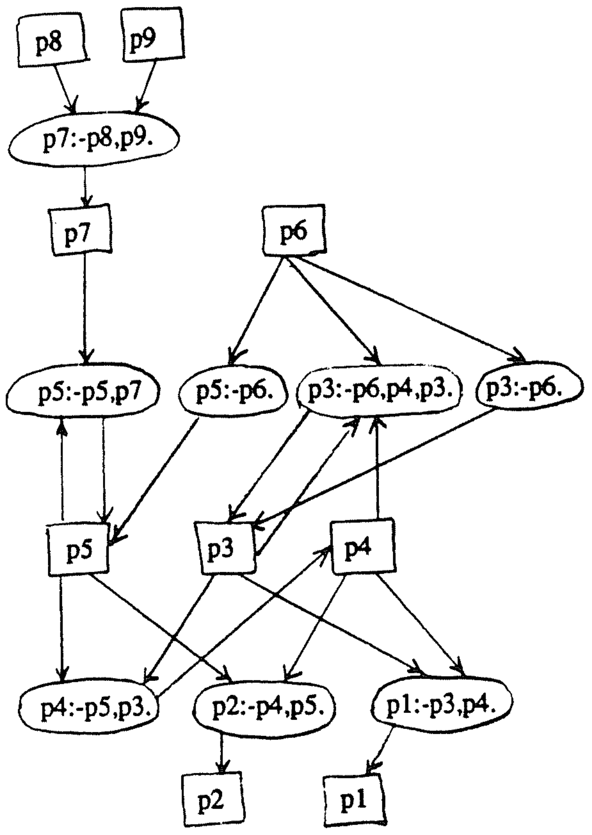
r: p:- p1,p2,...,pn

in the intensional database, then there is an arc going from node r to node p, and for each predicate pi there is an arc from node pi to node r.

Here is an example of an intensional database. For the sake of simplicity, we have omitted the variables in the rules:

```
r1    p1 :- p3,p4
r2    p2 :- p4,p5
r3    p3 :- p6,p4,p3
r4    p4 :- p5,p3
r5    p3 :- p6
r6    p5 :- p5,p7
r7    p5 :- p6
r8    p7 :- p8,p9
```

The rule/goal graph is:



2.5. Recursion

Recursion is often discussed in the single rule context. For the purpose of clarity and simplicity, let us first give some temporary definitions in this context. We say that a rule is recursive if it is of the form

p(t) :- ....,p(t'),.... .

For instance the rule

ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).

is recursive.

An interesting subcase is that of linear rules. Linear rules play an important role because (i) there is a belief that most "real life" recursive rules are indeed linear, and (ii) algorithms have been developed to handle them efficiently.

We say that a rule is linear if it is recursive, and the recursive predicate appears once and only once on the right. This property is sometime referred to as regularity (Chang [1981]). We believe the term linear to be more appropriate, and we think that regularity should be kept for another concept (which is not defined here).

For instance the rule:

sg(X,Y) :- p(X,XP),p(Y,YP),sg(XP,YP).

is linear, while the rule

ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).

is not.

These definitions are fairly simple in the single rule context. They are a little more involved in the context of a set of rules where properties have to be attached to predicates instead of rules. Consider the following database:

```
p(X,Y) :- b1(X,Z),q(Z,Y).
q(X,Y) :- p(X,Z),b2(Z,Y).
```

Neither of the rules are recursive according to the above definition, while clearly both predicates p and q are recursive.

We now come to the general definitions of recursion in the multirule context. Let p and q be two predicates. We say that p *derives* q (denoted p → q) if it occurs in the body of a rule whose head predicate is q. We define →+ to be the transitive closure (*not* the reflexive transitive closure) of →. A predicate p is said to be *recursive* if p →+ p. Two predicates p and q are *mutually recursive* if p →+ q and q →+ p. It can be easily shown that mutual recursion is an equivalence relation on the set of recursive predicates. Therefore the set of recursive predicates can be decomposed into disjoint blocs of mutually recursive predicates.

Given a set of rules, we say that the rule
p :- p1,p2,...,pn is *recursive* iff there exists pi in the body of the rule which is mutually recursive to p.

A recursive rule p :- p1,p2,...,pn is *linear* if there is one and only one pi in the body of the rule which is mutually recursive to p. A set of rules is *linear* if every recursive rule in it is linear. For instance, the following system is linear:

```
r1   p(X,Y) :- p1(X,Z),q(Z,Y).
r2   q(X,Y) :- p(X,Z),p2(Z,Y).
r3   p(X,Y) :- b3(X,Y).
r4   p1(X,Y) :- b1(X,Z),p1(Z,Y).
r5   p1(X,Y) :- b4(X,Y).
r6   p2(X,Y) :- b2(X,Z),p2(Z,Y).
r7   p2(X,Y) :- b5(X,Y).
```

The set of recursive predicates is {p,q,p1,p2}, the set of base predicates is {b1,b2,b3,b4,b5}. The blocks of mutually recursive predicates are {[p,q],[p1],[p2]}. The recursive rules are r1, r2, r4 and r6, and the system is linear even though rules r1 and r2 both have two recursive predicates on their right.

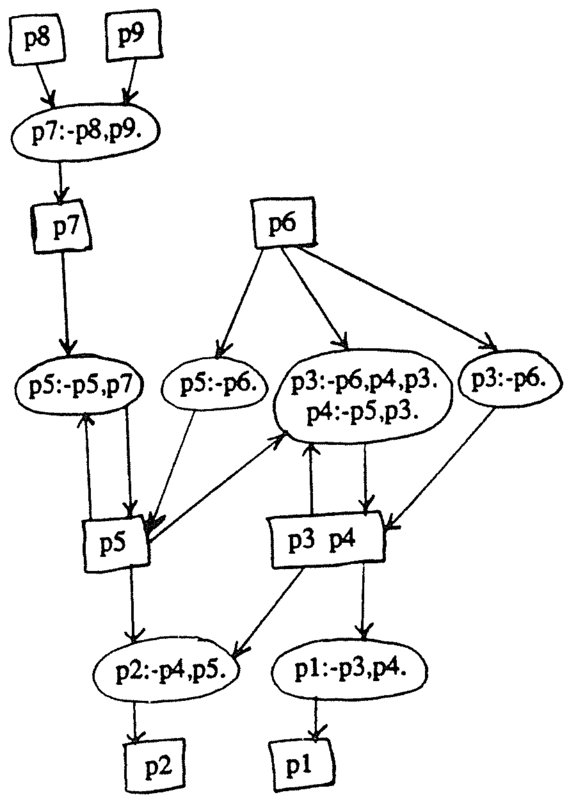
We say that two recursive rules are mutually recursive iff the predicates in their heads are mutually recursive. This defines an equivalence relation among the recursive rules.

Thus mutual recursion defines an equivalence class among recursive predicates and among the recursive rules, (Bancilhon [1985]). Therefore, it groups together all predicates which are mutually recursive to one another, i.e which must be evaluated as a whole. It also groups together all the rules which participate in evaluating those blocks of predicates. Let us now see how this can be represented in the rule/goal graph. We define the *reduced rule/goal graph* as follows:

Square nodes are associated with non-recursive predicates or with blocks of mutually recursive predicates and, oval nodes are associated with non-recursive rules or with blocks of mutually recursive rules. The graph essentially describes the non-recursive part of the database by grouping together all the predicates which are mutually recursive to one another and isolating the recursive parts. For every non-recursive rule of the form r: p:- p1,p2,...,pn, there is an arc going from node r to node p (if p is non-recursive), or to node [p], which is the node representing the set of predicates mutually recursive to p (if p is recursive). For each non-recursive predicate pi, there is an arc from the node pi to the node r, and for each recursive predicate pj there is an arc going from [pj] the node representing the set of predicates mutually recursive to pj.

Finally, each bloc of recursive rules [r] is uniquely associated to a set of mutually recursive predicates [p], and we draw an arc from [p] to [r] and an arc from [r] to [p]. We also draw an arc from q (if q is non-recursive) or from [q] (if q is recursive) to [r] if there is a rule in [r] which has q in its body. This grouping of recursive predicates in blocks of strongly connected components is presented in Morris et al. [1986].

Here is the representation of the previous database:



2.6. Safety of Queries

Given a query q in a database D, we say that q is *safe* in D if the answer to q is finite. Obviously unsafe queries are highly undesirable.

Sources of unsafeness are of two kinds

(i) the evaluable arithmetic predicates are interpreted by infinite tables. Therefore they are unsafe by definition. For instance the query greater-than(27,X) is unsafe.

(ii) rules with free variables in the head which do not appear in the body are a source of unsafeness in the presence of evaluable arithmetic predicates (the arithmetic predicates provide an infinite underlying domain, and the variable from the head of the rule which does not appear on the right ranges over that domain). Thus for instance, in the system

```
good-salary(X) :- X > 100000.
like(X,Y) :- nice(X).
nice(john).
```

the query like(john,X)? is unsafe because, in the minimal model of the database like(john,x) is true for every integer x. Note that if the first rule was not there, like(john,X)? would be safe and have answer like(john,john).

The problem of safety has received a lot of attention recently (Afrati et al. [1986], Kifer and Lozinskii [1987], Krishnamurthy et al. [1988], Ramakrishnan et al. [1987], Ullman [1985], Ullman and Van Gelder [1985], Van Gelder and Topor [1987], Zaniolo [1986]). We shall not survey those results here but merely present some simple sufficient syntactic conditions to guarantee safety. A rule is *range restricted* if every variable of the head appears somewhere in the body. Thus in this system:

```
r1    loves(X,Y) :- nice(X).
r2    loves(X,Y) :- nice(X),human(Y).
```

r1, which corresponds to "nice people love everything", is not range restricted while r2, which corresponds to "nice people love all humans", is. Obviously, every ground rule which is not a fact is not range restricted. For instance

loves(john,X).

is not range restricted.

A set of rules is range restricted if every rule in this set is range restricted.

It is known (Reiter [1978]) that if each evaluable predicate has a finite natural interpretation, and if the set of rules is range restricted, then every query defined over this set of rules is safe. This applies obviously to the case where there are no evaluable predicates. However, if there are evaluable predicates with infinite natural interpretations, safety is no longer assured. We now present a simple sufficient condition for safety in the presence of such predicates.

A rule is *strongly safe* iff: (1) it is range restricted, and (2) every variable in an evaluable predicate term also appears in at least one base predicate.

For example, the rule

well-paid(X) :- has-salary(X,Y), Y > 100K.

is strongly safe, whereas

great-salary(X) :- X > 100K.

is not strongly safe.

A set of rules is strongly safe if every rule in this set is strongly safe.

Any query defined over a set of strongly safe rules is safe. However, while this is a sufficient condition, it is not necessary. We can develop better conditions for testing safety, or leave it to the user to ensure that his queries are safe.

2.7. Effective Computability.

Safety, in general, does not guarantee that the query can be effectively computed. Consider for instance:

p1(1,X,Y) :- X≥Y.
p2(X,Y,2) :- X≤Y.
p(X,Y) :- p1(X,Z,Z),p2(Z,Z,Y).

The query p(X,Y) is safe (the answer is {p(1,2)}), but there is no general evaluation strategy which will compute the answer and terminate.

However, strongly safe rules are guaranteed to be safe *and* effectively computable.

In fact, while we might often be willing to let the user ensure that his queries are safe, it is desirable to ensure that the query can be computed without materializing "infinite" intermediate results. We now present a sufficient condition for ensuring this.

We first need some information about the way arithmetic predicates can propagate bindings. So we characterize each arithmetic predicate by a set of *finiteness constraints* (Zaniolo [1986], Ramakrishnan et al. [1987]). A finiteness constraint is a couple (X → Y) where X is a set of attributes and Y is a set of attributes. It is to be interpreted intuitively as "if the values of the X attributes are fixed then there is a finite number of values of the Y attributes associated with them". Therefore, while their semantics is different from that of functional dependencies, they behave in the same fashion (and have the same axiomatization). Of course, we assume that the natural interpretation of the evaluable predicate satisfies the set of finiteness constraints.

For instance, the ternary arithmetic predicate "sum" has the finiteness constraints:

$\{1,2\} \rightarrow \{3\}$
$\{1,3\} \rightarrow \{2\}$
$\{2,3\} \rightarrow \{1\}$

while the arithmetic predicate "greater than" has only trivial finiteness constraints.

Now consider a rule, and define each variable in the body to be *secure* if it appears in a non-evaluable predicate in the body or if it appears in position i in an evaluable predicate p and there is a subset I of the variables of p which are secure and $I \rightarrow \{i\}$. Note that the definition is recursive.

A rule is *bottom-up evaluable* if

1. it is range restricted, and
2. every variable in the body is secure.

For instance:

p(X,Y) :- Y=X+1, X=Y1+Y2, p(Y1,Y2).

is bottom-up evaluable because (i) Y1 and Y2 are secure (they appear in p which is non-evaluable), (ii) in X=Y1+Y2, the finiteness constraint $\{Y1,Y2\} \rightarrow \{X\}$ holds, therefore X is secure, and (iii) in Y=X+1, the finiteness constraint $\{X\} \rightarrow \{Y\}$ holds, therefore Y is secure.

On the contrary

p(X,Y) :- X>Y1, q(Y1,Y).

is not bottom-up evaluable because X is not secure.

A set of rules is bottom-up evaluable if every rule in this set is bottom-up evaluable.

Any computation using only a set of bottom-up evaluable rules can be carried out without materializing infinite intermediate results. The computation proceeds in a strictly bottom-up manner, using values for the body variables to produce values for the head variables. The bottom-up evaluability criterion ensures that the set of values for body variables is finite at each step. However, there may be an infinite number of steps. For example, if we repeatedly apply the bottom-up evaluable rule given above, at each step we have a finite number of values (in this case, a unique value) for Y1 and Y2, and hence for X and Y. However, we can apply the rule an infinite number of times, producing new values for X and Y at each step.

3. Classification of the Strategies

In the past five years, a large number of strategies to deal with Horn rules have been presented in the literature. A strategy is defined by (i) an application domain (i.e., a class of rules for which it applies) and (ii) an algorithm for replying to queries given such a set of rules.

In studying the strategies, we found that the methods were described at different levels of detail and using different formalisms, that they were sometimes very difficult to understand (and sometimes were understood differently by subsequent authors), that the application domain was not always very clearly defined, and that no performance evaluation was given for any of the strategies, which left the choice of a given strategy completely open when the application domain was the same. Finally, we found that some of the strategies were in fact the same.

We think that the strategies should be compared according to the following criteria (i) size of the application domain, (the larger the better), (ii) performance of the strategy, (the faster the better) and (iii) ease of implementation (the simpler the better). While the last criterion is somehow subjective, the first two should be quantifiable. In this section, we give a complete description of our understanding of the strategies and of their application domains, and we demonstrate each one of them through an example. As much as possible, we have tried to use the same example, except for some "specialized" strategies where we have picked a specific example which exhibits its typical behavior.

3.1. Characteristics of the Strategies

3.1.1. Query Evaluation vs. Query Optimization

Let us first distinguish between two approaches: one first class of strategies consists of an actual query evaluation algorithm, i.e. a program which, given a query and a database, will produce the answer to the query. We will call these *methods*. Representatives of this class are: *Henschen-Naqvi, Query/Subquery (QSQ)* or *Extension Table, APEX, Prolog, Naive Evaluation* and *Semi-Naive Evaluation.*

The strategies in the second class assume an underlying simple strategy (which is in fact naive or semi-naive evaluation) and optimize the rules to make their evaluation more efficient. They can all be described as *term rewriting systems*. These include: *Aho-Ullman, Counting and Reverse Counting, Magic Sets, Generalized Magic Sets* and *Kifer-Lozinskii.*

Note that this distinction is somehow arbitrary: each of the optimization strategies could be described as a method (when adding to it naive or semi-naive evaluation). However, this decomposition has two advantages: (i) it *might* make sense from an implementation point of view to realize the optimization strategies as term rewriting systems on top of an underlying simpler method such as naive evaluation, and (ii) from a pedagogical standpoint, they are much easier to understand this way, because presenting them as term rewriting systems indeed captures their essence.

The subsequent characteristics only relate to pure methods.

3.1.2. Interpretation vs. Compilation

A method can be *interpreted* or *compiled*. The notion is somehow fuzzy, and difficult to characterize formally. We say that the strategy is compiled if it consists of two phases: (i) a compilation phase, which accesses only the intensional database, and which generates an "object program" of some form, and (ii) an execution phase, which executes the object program against the facts only. A second characteristic of compiled methods is that all the database query forms (i.e., the query forms on base relations which are directly sent to the DBMS) are generated during the compilation phase. This condition is very important, because it allows the DBMS to precompile the the query forms. Otherwise the database query forms are repetitively compiled by the DBMS during the execution of the query, which is a time consuming operation. If these two conditions do not hold, we say that the strategy is interpreted. In this case, no object code is produced and there is a fixed program, the "interpreter", which runs against the query, the set of rules and the set of facts.

3.1.3. Recursion vs. Iteration

A rule processing strategy can be *recursive* or *iterative*. It is iterative if the "target program" (in case of a compiled approach) or the "interpreter" (in case of the interpreted approach) is iterative. It is recursive if this program is recursive, i.e., uses a stack as a control mechanism. Note that in the iterative methods, the data we deal with is statically determined. For instance, if we use temporary relations to store intermediate results, there are a finite number of such temporary relations. On the contrary, in recursive methods the number of temporary relations maintained by the system is unbounded.

3.1.4. Potentially Relevant Facts

Let D be a database and q be a query. A fact p(a) is *relevant* to the query iff there exists a derivation p(a) $\rightarrow^*$ q(b) for some b in the answer set. The notion of relevant fact was introduced in Lozinskii [1985], we use it here with a somewhat different meaning. If we know all the relevant facts in advance, instead of using the database to reply to the query, we can use the relevant part of the database only, thus cutting down on the set of facts to be processed. A *sufficient set of relevant facts* is a set of facts such that replacing the database by this set of facts gives the same answer to the query. Unfortunately, in general there does not exist a unique minimal set of facts as the following example shows:

```
suspect(X) :- long-hair(X).
suspect(X) :- alien(X).
long-hair(antoine).
alien(antoine).
```

Minimal sets of facts with respect to the query suspect(X)? are {long-hair(antoine)} and {alien(antoine)}. The second unfortunate thing about relevant facts is that it is in general impossible to find all the relevant facts in advance without spending as much effort as in replying to the query. Thus, all methods have a way of finding a super-set of relevant facts. We call this set the *set of potentially relevant facts*. A set of potentially relevant facts is *valid* if it contains a sufficient set of relevant facts. An obvious but not very interesting valid set is the set of all facts of the database.

3.1.5. Top Down vs. Bottom Up

Consider the following set of rules and the query:

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
query(X) :- ancestor(john,X).
```

We can view each of these rules as productions in a grammar. In this context, the database predicates (parent in this example) appear as terminal symbols, and the derived predicates (ancestor in this example) appear as the non-terminal symbols. Finally, to pursue the analogy, we shall take the distinguished symbol to be query(X). Of course, we know that the analogy does not hold totally, for two reasons: (i) the presence of variables and constants in the literals and (ii) the lack of order between the literals of a rule (for instance "parent(X,Z), ancestor(Z,Y)" and "ancestor(Z,Y), parent(X,Z)" have the same meaning). But we shall ignore these differences, and use the analogy informally.

Let us now consider the language generated by this "grammar". It consists of

```
{parent(john,X);
 parent(john,X),parent(X,X1);
 parent(john,X),parent(X,X1),parent(X1,X2);
 ...}
```

This language has two interesting properties: (i) it consists of first order sentences involving only base predicates, i.e., each word of this language can be directly evaluated against the database, and (ii) if we evaluate each word of this language against the database and take the union of all these results, we get the answer to the query.

There is a minor problem here: the language is not finite, and we would have to evaluate an infinite number of first order sentences. To get out of this difficulty, we use termination conditions which tell us when to stop. An example of such a termination condition is: if one word of the language evaluates to the empty set, then all the subsequent words will also evaluate to the empty set, so we can stop generating new words. Another example of a termination condition is: if a word evaluates to a set of tuples, and all these tuples are already in the evaluation of the words preceding it, then no new tuple will ever be produced by the evaluation of any subsequent word, thus we can stop at this point.

All query evaluation methods in fact do the following:

(i) generate the language, (ii) while the language is generated, evaluate all its sentences and (iii) at each step, check for the termination condition.

Therefore, there are essentially two classes of methods: those which generate the language bottom up, and those which generate the language top-down. The bottom-up strategies start from the terminals (i.e., the base relations) and keep assembling them to produce non-terminals (i.e derived relations) until they generate the distinguished symbol (i.e., the query). The top-down strategies start from the distinguished symbol (the query) and keep expanding it by applying the rules to the non-terminals (derived relations). As we shall see, top-down strategies are often more efficient because they "know" which query is being solved, but they are more complex. Bottom up strategies are simpler, but they compute a lot of useless results because they do not know what query they are evaluating.

4. The Methods

We shall use the same example for most of the methods. The intensional database and query are:

```
R1   ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
R2   ancestor(X,Y) :- parent(X,Y).
R3   query(X) :- ancestor(aa,X).
```

The extensional database is:

```
parent(a,aa).
parent(a,ab).
parent(aa,aaa).
parent(aa,aab).
parent(aaa,aaaa).
parent(c,ca).
```

4.1. Naive Evaluation

Naive Evaluation is a bottom-up, compiled, iterative strategy.

Its application domain is the set of bottom-up evaluable rules.

In a first phase, the rules which derive the query are compiled into an iterative program. The compilation process uses the reduced rule/goal graph. It first selects all the rules which derive the query. A temporary relation is assigned to each derived predicate in this set of rules. A statement which computes the value of the output predicate from the value of the input predicates is associated with each rule node in the graph. With each set of mutually recursive rules, there is associated a loop which applies the rules in that set until no new tuple is generated. Each temporary relation is initialized to the empty set. Then computation proceeds from the base predicates capturing the nodes of the graph.

In this example, the rules which derive the query are {R1, R2, R3}, and there are two temporary relations: ancestor and query. The method consists in applying R2 to parent, producing a new value for ancestor, then applying R1 to ancestor until no new tuple is generated, then applying R3.

The object program is:

```
begin
initialize ancestor to the empty set;
evaluate (ancestor(X,Y) :- parent(X,Y));
insert the result in ancestor;
while "new tuples are generated" do
  begin
  evaluate (ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)) using the current value of ancestor;
  insert the result in ancestor
  end;
evaluate (query(X) :- ancestor(aa,X));
insert the result in query
end.
```

The execution of the program against the data goes as follows:

Step 1: Apply R2.
The resulting state is:
ancestor = {(a,aa), (a,ab), (aa,aaa), (aa,aab), (aaa,aaaa), (c,ca)}
query = {}

Step 2: Apply R1.
The following new tuples are generated:
ancestor: {(a,aaa), (a,aab), (aa,aaaa)}
And the resulting state is:
ancestor = {(a,aa), (a,ab), (aa,aaa), (aa,aab), (aaa,aaaa), (c,ca), (a,aaa), (a,aab), (aa,aaaa)}
query = {}

New tuples have been generated so we continue:

Step 3: Apply R1.
The following tuples are generated:
ancestor: {(a,aaa), (a,aab), (aa,aaaa), (a,aaaa)}
The new state is:

ancestor = {(a,aa), (a,ab), (aa,aaa), (aa,aab), (aaa,aaaa), (c,ca), (a,aaa), (a,aab), (aa,aaaa), (a,aaaa)}
query = {}

Because (a,aaaa) is new, we continue:

Step 4: Apply R1.
The following tuples are generated:
ancestor: {(a,aaa), (a,aab), (aa,aaaa), (a,aaaa)}

Because there are no new tuples, the state does not change and we move to R3.

Step 5: Apply R3.
The following tuples are produced:
query: {(aa,aaa), (aa,aaaa)}
The new state is:
ancestor = {(a,aa), (a,ab), (aa,aaa), (aa,aab), (aaa,aaaa), (c,ca), (a,aaa), (a,aab), (aa,aaaa), (a,aaaa)}
query = {(aa,aaa), (aa,aaaa), (aa,aab)}.

The algorithm terminates.

In this example, we note the following problems: (i) the entire relation is evaluated, i.e., the set of potentially relevant facts is the set of facts of the base predicates which derive the query, and (ii) step 3 completely duplicates step 2.

Naive evaluation is the most widely described method in the literature. It has been presented in a number of papers under different forms. The inference engine of SNIP, presented in McKay and Shapiro [1981], is in fact an interpreted version of naive evaluation. The method described in Chang [1981], while based on a very interesting language paradigm and restricted to linear systems, is a compiled version of naive evaluation based on relational algebra. The method in Marque-Pucheu [1983] is a compiled version of naive evaluation using a different algebra of relations. The method in Bayer [1985] is another description of naive evaluation. The framework presented in Delobel [1986] also uses naive evaluation as its inference strategy. SNIP is, to our knowledge, the only existing implementation in the general case.

4.2. Semi-Naive Evaluation

Semi-naive evaluation is a bottom-up, compiled and iterative strategy.

Its application range is the set of bottom-up evaluable rules.

This method uses the same approach as naive evaluation, but tries to cut down on the number of duplications. It behaves exactly as naive evaluation, except for the loop mechanism where it tries to be smarter.

Let us first try to give an idea of the method as an extension of naive evaluation. Let p be a recursive predicate; consider a recursive rule having p as a head predicate and let us write this rule:

$$p :- \phi(p1,p2,...,pn,q1,q2,...,qm).$$

where ϕ is a first order formula, p1,p2,...,pn are mutually recursive to p, and q1,q2,..,qm are base or derived predicates, which are not mutually recursive to p.

In the naive evaluation strategy, all the qi's are fully evaluated when we start computing p and the pi's. On the other hand p and the pi's are all evaluated inside the same loop (together with the rest of predicates mutually recursive to p).

Let pj(i) be the value of the predicate pj at the i-th iteration of the loop. At this iteration, we compute

$$\phi(p1(i),p2(i),...,pn(i),q1,q2,...,qm).$$

During that same iteration each pj receives a set of new tuples. Let us call this new set dpj(i). Thus the value of pj at the beginning of step (i+1) is pj(i) + dpj(i) (where + denotes union).

At step (i+1) we evaluate

$$\phi((p1(i)+dp1(i)),...,(pn(i)+dpn(i)),q1,...,qm),$$

which, of course, recomputes the previous expression (because ϕ is monotonic).

The ideal however, is to compute only the *new* tuples i.e the expression:

$$d\phi(p1(i),dp1(i),\ldots,pn(i),dpn(i),q1,\ldots,qm) = \phi((p1(i)+dp1(i)),\ldots,(pn(i)+dpn(i)),q1,\ldots,qm) - \phi(p1(i),\ldots,pn(i),q1,\ldots,qm)$$

The basic principle of the semi-naive method is the evaluation of the differential of ϕ instead of the entire ϕ at each step. The problem is to come up with a first order expression for $d\phi$, which does not contain any difference operator. Let us assume there is such an expression, and describe the algorithm. With each recursive predicate p are associated four temporary relations p.before, p.after, dp.before and dp.after. The object program for a loop is as follows:

```
while "the state changes" do
  begin
  for all mutually recursive predicates p do
    begin
    initialize dp.after to the empty set;
    initialize p.after to p.before;
    end
  for each mutually recursive rule do
    begin
    evaluate $d\phi(p1,dp1,\ldots,pn,dpn,q1,\ldots,qn)$ using the current values of
    pi.before for pi and of dpi.before for dpi;
    add the resulting tuples to dp.after;
    add the resulting tuples to p.after
    end
  end.
```

All we have to do now is provide a way to generate $d\phi$ from ϕ. The problem is not solved in its entirety and only a number of transformations are known. In Bancilhon [1985], some of them are given in terms of relational algebra.

It should be noted however, that for the method to work, the only property we have to guarantee is that:

$$\phi(p1+dp1,\ldots) - \phi(p1,\ldots) \subseteq d\phi(p1,dp1,\ldots) \subseteq \phi(p1+dp1,\ldots)$$

Clearly, the closer $d\phi(p1,dp1,\ldots)$ is to $(\phi(p1+dp1,\ldots) - \phi(p1,\ldots))$, the better the optimization is. In the worse case, where we use ϕ for $d\phi$, semi-naive evaluation behaves as naive evaluation. Here are some simple examples of rewrite rules:

if $\phi(p,q) = p(X,Y),q(Y,Z)$, **then** $d\phi(p,dp,q) = dp(X,Y),q(Y,Z)$

More generally when ϕ is linear in p, the expression for $d\phi$ is obtained by replacing p by dp.

if $\phi(p1,p2) = p1(X,Y),p2(Y,Z)$,
then $d\phi(p,dp) = p1(X,Y),dp2(Y,Z)+dp1(X,Y),p2(Y,Z)+dp1(X,Y),dp2(Y,Z)$

Note that this is not an exact differential but a reasonable approximation.

The idea of semi-naive evaluation underlies many papers. A complete description of the method based on relational algebra is given in Bancilhon [1985]. The idea is also present in Bayer [1985].

It should also be pointed out that, in the particular case of linear rules, because the differential of $\phi(p)$ is simply $\phi(dp)$, it is sufficient to have an inference engine which only uses the new tuples. Therefore many methods which are restricted to linear rules do indeed use semi-naive evaluation. Note also that when the rules are not linear, applying naive evaluation only to the "new tuples" is an incorrect method (in the sense that it does not produce the whole answer to the query). This can be easily checked on the recursive rule:

ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).

In this case, if we only feed the new tuples at the next stage, the relation which we compute consists of the ancestors whose distance to one another is a power of two.

To our knowledge, outside of the special case of linear rules, the method as a whole has not been implemented.

4.3. Iterative Query/Subquery

Iterative Query/Subquery (QSQI) is an interpreted, top-down strategy.

Its application domain is the set of range restricted rules without evaluable predicates.

The method associates a temporary relation with every relation which derives the query, but the computation of the predicates deriving the query is done at run time. QSQI also stores a set of queries which are currently being evaluated. When several queries correspond to the same query form, QSQI stores and executes them as a single object. For instance, if we have the queries p(a,X) and query p(b,X), we can view this as query p({a,b},X). We call such an object a *generalized query.* The state memorized by the algorithm is a couple <Q,R>, where Q is a set of generalized queries, and R is a set of derived relations, together with their current values.

The iterative interpreter is as follows:

```
Initial state is <{query(X)},{}>
while the state changes do
 for all generalized queries in Q do
   for all rules whose head matches the generalized query do
     begin
     unify rule with the generalized query;
     (i.e propagate the constants. this generates new generalized queries for
     each derived predicate in the body by looking up the base relations.)
     generate new tuples;
     (by replacing each base predicate on the right by its value and every
     derived predicate by its current temporary value.)
     add these new tuples to R;
     add these new generalized queries to Q
     end
```

Let us now run this interpreter against our example logic database:

The initial state is: <{query(X)},{}>

Step 1

We try to solve query(X). Only rule R3 applies. The unification produces the generalized query ancestor({aa},X). This generates temporary relations for query and ancestor with empty set values. Attempts at generating tuples for this generalized query fail.

The new state vector is:

<{query(X),ancestor(aa,X)}, {ancestor={},query={}}>

Step 2

A new generalized query has been generated, so we go on. We try to evaluate each of the generalized queries: query(X) does not give anything new, so we try ancestor({aa},X).
Using rule R2, and unifying, we get parent(aa,X). This is a base relation, so we can produce a set of tuples. Thus we generate a value for ancestor which contains all the tuples of parent(aa,X) and the new state vector is:

<{query(X),ancestor(aa,X)}, {ancestor={(aa,aaa),(aa,aab)},query={}}>

We now solve ancestor(aa,X) using R1. Unification produces the expression :

parent(aa,Z),ancestor(Z,Y).

We try to generate new tuples from this expansion and the current ancestor value but get no tuples. We also generate new generalized queries by looking up parent and instantiating Z. This produces the new expression:

parent(aa,{aaa,aab}),ancestor({aaa,aab},Z).

This creates two new queries which are added to the generalized query and the new state is:

<{query(X),ancestor({aa,aaa,aab},X)}, {ancestor={(aa,aaa),(aa,aab)},query={}}>

Step 3

New generalized queries and new tuples have been generated so we continue. We first solve query(X) using R3 and get the value {(aa,aaa), (aa,aab)} for query. The resulting new state is:

<{query(X),ancestor({aa,aaa,aab},X)}, {ancestor={(aa,aaa),(aa,aab)}, query={(aa,aaa),(aa,aab)}}>

We now try to solve ancestor({aa,aaa,aab},X). Using R2, we get parent({aa,aaa,aab),X) which is a base relation and generates the following tuples in ancestor: {(aa,aaa),(aa,aab),(aaa,aaaa)}. This produces the new state:

<{query(X),ancestor({aa,aaa,aab},X)}, {ancestor={(aa,aaa),(aa,aab),(aaa,aaaa)},
    query={(aa,aaa),(aa,aab)}}>

We now solve ancestor({aa,aaa,aab},X)} using R1 and we get: parent({aa,aaa,aab},Z),ancestor(Z,Y). We bind Z by going to the parent relation, and we get: parent({aa,aaa,aab},{aaa,aab,aaaa}), ancestor({aaa,aab,aaaa},Y). This generates the new generalized query ancestor({aaa,aab,aaaa},Y) and the new state:

<{query(X),ancestor({aa,aaa,aab,aaaa},X)}, {ancestor={(aa,aaa),(aa,aab),(aa,aaaa),(aaa,aaaa)},
    query={(aa,aaa),(aa,aaaa),(aa,aab)}}>

Step 4

A new generalized query has been generated, so we continue. Solving the ancestor queries using R2 will not produce any new tuples, and solving it with R3 will not produce any new generalized query nor any tuples. The algorithm terminates.

Concerning the performance of the method, one can note that (i) the set of potentially relevant facts is better than for naive (in this example it is optimal), and (ii) QSQI has the same duplication problem as naive evaluation: each step entirely duplicates the previous strategy.

Iterative Query/Subquery is presented in Vieille [1986]. To our knowledge it has not been implemented.

4.4. Recursive Query/Subquery or Extension Tables

Recursive Query/Subquery (QSQR) is a top-down interpreted recursive strategy.

The application domain is the set of range restricted rules without evaluable predicates.

It is of course a recursive version of the previous strategy. As before, we maintain temporary values of derived relations and a set of generalized queries. The state memorized by the algorithm is still a couple <Q,R>, where Q is a set of generalized queries and R is a set of derived relations together with their current values. The algorithm uses a selection function which, given a rule, can choose the first and the next derived predicate in the body to be "solved".

The recursive interpreter is as follows:

```
procedure evaluate(q)  (* q is a generalized query *)
begin
while "new tuples are generated" do
  for all rules whose head matches the generalized query do
    begin
    unify the rule with the generalized query; (i.e., propagate the constants)
    until there are no more derived predicates on the right do
      begin
```

```
        choose the first/next derived predicate according to the selection function;
        generate the corresponding generalized query;
        (This is done by replacing in the rule each base predicate by its value
        and each previously solved derived predicate by its current value).
        eliminate from that generalized query the queries that are already in Q;
        this produces a new generalized query q';
        add q' to Q;
        evaluate(q')
        end;
      replace each evaluated predicate by its value and evaluate the generalized query q;
      (This can be done in some order without waiting for all predicates to be evaluated.)
      add the results in R;
      return the results
      end
end.
Initial state is <{query(X)},{}>
evaluate(query(X)).
```

It is important to note that this version of QSQ is very similar to Prolog. It solves goals in a top-down fashion using recursion, and it considers the literals ordered in the rule (the order is defined by the selection function). The important differences with Prolog are: (i) the method is set-at-a-time instead of tuple-at-a-time, through the generalized query concept, and (ii) as pointed out in Dietrich and Warren [1985], the method uses a dynamic programming approach of storing the intermediate results and re-using them when needed. This dynamic programming feature also solves the problem of cycles in the facts: while Prolog will run in an infinite loop in the presence of such cycles, QSQR will detect them and stop the computation when no new tuple is generated. Thus, QSQR is complete over its application domain whereas Prolog is not.

Here is the ancestor example:

```
evaluate(query(X))
  use rule R3
  query(X) :- ancestor(aa,X)
  this generates the query ancestor({aa},X)
  new state is: <{ancestor({aa},X), query(X)},{}>
  evaluate(ancestor({aa},X)
    Step 1 of the iteration
    use rule R1
    ancestor({aa},Y) :- parent({aa},Z), ancestor(Z,Y).
    by looking up parent we get the bindings {aaa,aab} for Z.
     this generates the query ancestor({aaa,aab},X)
    new state is: <{ancestor({aa,aaa,aab},X), query(X)},{}>
    evaluate (ancestor({aaa,aab},X))
    (this is a recursive call)
      Step 1.1
      use R1
      ancestor({aaa,aab},Y) :- parent({aaa,aab},Z),ancestor(Z,Y).
      by looking up parent we get the binding {aaaa} for Z
      new state is: <{ancestor({aa,aaa,aab,aaaa},X), query(X)},{}>
      evaluate(ancestor({aaaa},X))
      (this is a recursive call)
        Step 1.1.1
        use R1
        ancestor({aaaa},Y) :- parent({aaaa},Z),ancestor(Z,Y).
        by looking up parent we get no binding for Z
        use R2
```

```
            ancestor({aaaa},Y) :- parent({aaaa},Y)
            this fails to return any tuple
            end of evaluate(ancestor({aaaa},X))
            Step 1.1.2
            nothing new is produced
            end of evaluate(ancestor({aaaa},Y))
          use R2
          ancestor({aaa,aab},Y) :- parent({aaa,aab},Y)
          this returns the tuple ancestor(aaa,aaaa)
          new state is: <{ancestor({aa,aaa,aab,aaaa},X),
          query(X)}, {ancestor={(aaa,aaaa)}}>
          Step 1.2
          same as Step 1, nothing new produced
          end of evaluate (ancestor({aaa,aab},X))
           (popping from the recursion in rule R1, we have:)
           a new tuple generated - ancestor(aa,aaaa)
          new state is: <{ancestor({aa,aaa,aab,aaaa},X), query(X)},
                        {ancestor={(aaa,aaaa), (aa,aaaa)}}>
        use rule R2
        ancestor({aa},X) :- parent({aa},Y)
        returns the tuples ancestor(aa,aaa) and ancestor(aa,aab)
        new state is: <{ancestor({aa,aaa,aab,aaaa},X), query(X)},{ancestor={(aaa,aaaa),(aa,aaaa),(aa,aaa),(aa,aab)}}>
        Step 2
        nothing new produced
        end of evaluate({aa},X)
      generate tuples from R3
      new state is: <{ancestor({aa,aaa,aab,aaaa},X), query(X)},{ancestor={(aaa,aaaa),
                  (aa,aaaa),(aa,aaa),(aa,aab)},query=(aa,aaaa), (aa,aaa),(aa,aab)}}>
      end of evaluate(query(X))
```

Recursive Query/Subquery is described in Vieille [1986]. A compiled version has been implemented on top of the INGRES relational system. In Dietrich and Warren [1985], along with a good survey of some of these strategies, a method called "extension tables" is presented. It is, up to a few details, the same method.

4.5. Henschen-Naqvi

Henschen-Naqvi is a top-down, compiled and iterative method.

The application domain is that of linear range restricted rules.

The method has a compilation phase which generates an iterative program. That iterative program is then run against the data base. The general strategy is fairly complex to understand, and we shall restrict ourselves to describing it in the "typical case" which is:

```
p(X,Y) :- up(X,XU),p(XU,YU),down(YU,Y).
p(X,Y) :- flat(X,Y).
query(X) :- p(a,X).
```

Note that the relation names *up* and *down* are not to be confused with the notions "top-down" or "bottom-up", which are characteristics of evaluation strategies. Let us introduce some simple notation, which will make reading the algorithm much simpler. Since we are only dealing with binary relations, we can view these as set-to-set mappings. Thus, the relation r associates with each set A a set B, consisting of all the elements related to A by r. We denote A.r the image of A by r, and we have:

$$A.r = \{ y \mid r(x,y) \text{ and } x \in A \}$$

If we view relations as mappings, we can compose them, and we shall denote r.s the composition of r and s. Therefore:

$$A.(r.s) = (A.r).s$$

This approach is similar to the formalism described in Gardarin and Maindreville [1986]. We shall denote the composition of relation r n times with itself r^n. Finally we shall denote set union by '+'. Once this notation is introduced, it is easy to see that the answer to the query is

$$\{a\}.flat + \{a\}.up.flat.down + \{a\}.up.up.flat.down.down + \ldots + \{a\}.up^n.flat.down^n + \ldots$$

The state memorized by the algorithm is a couple <V,E>, where V is a the value of a unary relation and E is an expression. At each step, using V and E, we compute some new tuples and compute the new values of V and E.

The iterative program is as follows:

```
V := {a};
E := λ;        /* the empty string */
while "new tuples are generated in V" do
   begin
   /* produce some answer tuples */
   answer := answer + V.flat.E;
   /* compute the new value */
   V := V.up ;
   /* compute the new expression */
   E := E | .down;
  end.
```

Note that E is an *expression*, and is augmented each time around the loop by concatenating ".down" to it through the "cons" operator. As can be seen from this program, at step i, the value V represents $\{a\}.up^i$ and the expression E represents $down^i$. Therefore the produced tuples are:

$$\{a\}.up^i.flat.down^i.$$

This is not meant to be a complete description of the method, but a description of its behavior in the typical case.

The Henschen-Naqvi method is described in Henschen and Naqvi [1984]. The method has been implemented in the case described here. This implementation can be found in Laskowski [1984]. An equivalent strategy is described using a different formalism in Gardarin and Maindreville [1986]. The performance of the strategy is compared to Semi-Naive evaluation and another method (not described here) in Han and Lu [1986].

4.6. Prolog

Prolog (Roussel [1975]) is a top-down, interpreted and recursive method.

The application domain of Prolog is difficult to state precisely: (i) it is data dependent in the sense that the facts have to be acyclic for the interpreter to terminate, and (ii) there is no simple syntactic characterization of a terminating Prolog program. The job of characterizing the "good" rules is left to the programmer.

We consider its execution model to be well known and will not describe it. In fact Prolog is a programming language and not a general strategy to evaluate Horn clauses. We essentially mention Prolog for the sake of completeness and because it is interesting to compare its performance to the other strategies.

4.7. APEX

APEX is a strategy which is difficult to categorize. It is partly compiled in the sense that a graph similar to the predicate connection graph is produced from the rules, which takes care of some of the preprocessing needed for interpretation. It is not fully compiled in the sense that the program which runs against the database is still unique (but driven by the graph). It is, however, clearly recursive, because the interpreter program is recursive. Finally, it is partly top-down and partly bottom-up as will be seen in the interpreter.

The application domain of APEX is the set of range restricted rules which contain no constants and no evaluable predicates.

The interpreter takes the form of a recursive procedure, which, given a query, produces a set of tuples for this query. It is as follows:

```
procedure solve(query,answer)
begin
answer := {};
if query q is on a base relation
then evaluate q against the date base
else
  begin
   select the relevant facts for q in the base predicates;
   put them in relevant;
   while new tuples are generated do
      begin
      for each rule do (this can be done in parallel)
         begin
         instantiate the right predicates with the relevant facts and produce tuples for the left predicate;
         add these tuples to the set of relevant facts;
         initialize the set of useful facts to the set of relevant facts;
         for each literal on the right do (this can be done in parallel)
            begin
            for each matching relevant fact do
               begin
               plug the fact in the rule and propagate the constants;
               this generates a new rule and a new set of queries;
               for all these new queries q' do
                  begin
                  solve(q',answer(q')) (this is the recursion step)
                  add answer(q') to the useful facts
                  end
               end
            instantiate the right predicates with the useful facts;
            produce tuples for the left predicate;
            add these to the relevant facts;
            extract the answer to q from the relevant facts
            end
          end
      end
    end.
  end;
  solve(query(X),answer).
```

Let us now run this program against our ancestor example. We cannot have a constant in the rules and we must modify our rule set and solve directly the query ancestor(aa,X):

solve (ancestor(aa,X), answer)
we first select the relevant base facts.
relevant = {parent(aa,aaa),parent(aa,aab)};
we now start the main iteration:
Step 1
rule R1
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)

```
we cannot produce any new tuple from this rule because ancestor
does not yet have any relevant fact
useful = {parent(aa,aaa),parent(aa,aab)};
process parent(X,Z)
   use parent(aa,aaa)
      the new rule is
      parent(aa,aaa),ancestor(aaa,Y)
      solve(ancestor(aaa,Y),answer1)
      ... (this call is not described)
      this returns
      {ancestor(aaa,aaaa)}, which we add to useful
      useful = {parent(aa,aaa),parent(aa,aab),ancestor(aaa,aaaa)};
   use parent(aa,aab)
      the new rule is
      parent(aa,aab),ancestor(aab,Y)
      solve(ancestor(aab,Y),answer2)
      ... (this call is not described)
      this returns nothing
process ancestor(Z,Y)
we instantiate parent and ancestor with the useful facts.
this produces ancestor(aa,aaaa)
we add it to the relevant facts:
relevant = {parent(aa,aaa),parent(aa,aab), ancestor(aa,aaaa)};

rule R2
   ancestor(X,Y) :- parent(X,Y)
   using the relevant facts we produce {ancestor(aa,aaa),ancestor(aa,aab)}
   we add these to relevant:
   relevant = {parent(aa,aaa),parent(aa,aab), ancestor(aa,aaa), ancestor(aa,aab), ancestor(aa,aaaa)};
   this rule does not produce any subquery

Step 2
   will not produce anything new,
   and so the algorithm stops.
```

The APEX method is described in Lozinskii [1985]. The method has been implemented.

5. The Optimization Strategies

We now turn to the description of the second class of strategies: the optimization strategies.

The main drawbacks of the naive evaluation method are:

1. The potential set of relevant facts is too large (i.e., it does not make good use of the query bindings), and
2. It generates a lot of duplicate computation.

A number of optimization strategies have recently been proposed to overcome these two difficulties.

5.1. Aho-Ullman

Aho and Ullman (Aho and Ullman [1979]) present an algorithm for optimizing recursive queries by commuting selections with the least fixpoint operator (LFP). The input is an expression

$$\sigma_F(LFP(r=f(r))$$

where f(r) is a monotonic relational algebra expression (under the ordering of set inclusion) and contains at most one occurrence of r. The output is an equivalent expression where the selection has been pushed through as far as possible.

We introduce their notation and ideas through an example. Consider:

a(X,Y) :- a(X,Z), p(Z,Y).

a(X,Y) :- p(X,Y).

q(X) :- a(john,X).

Aho-Ullman write this as:

$\sigma_{a_1=john}(LFP(a = a.p \cup p))$

In this definition, *a* is a relation which is defined by a *fixpoint* equation in relational algebra, and *p* is a base relation. If we start with *a* empty and repeatedly compute *a* using the rule a = a.p ∪ p, at some iteration, there is no change (since the relation *p* is finite). Because the function used in the fixpoint equation is monotonic, this is the *least fixpoint* of the fixpoint equation (Tarski [1955]). It is the smallest relation *a* which satisfies the equation, i.e. contains every tuple which can be generated by using the fixpoint rule, and no tuple which cannot. The query is simply the selection a_1=john applied to this relation. Thus, the query is a selection applied to the transitive closure of p.

We now describe how the Aho-Ullman algorithm optimizes this query. We use '.' to denote composition, which is a join followed by projecting out the join attributes. We begin with the expression

$\sigma_{a_1=john}(a)$

and by replacing a by f(a) we generate

$\sigma_{a_1=john}(a.p \cup p))$

By distributing the selection across the join, we get

$\sigma_{a_1=john}(a.p) \cup \sigma_{a_1=john}(p)$.

Since the selection in the first subexpression only involves the first attribute of a, we can rewrite it as

$\sigma_{a_1=john}(a) . p$

We observe that this contains the subexpression

$\sigma_{a_1=john}(a)$

which was the first expression in the series. If we denote this by E, the desired optimized expression is then

$LFP(E = E.p \cup \sigma_{a_1=john}(p))$

This is equivalent to the Horn Clause query:

a(john,Y) :- a(john,Z), p(Z,Y).
a(john,Y) :- p(john,Y).
q(X) :- a(john,X).

The essence of the strategy is to construct a series of equivalent expressions starting with the expression $\sigma_F(r)$ and repeatedly replacing the single occurrence of r by the expression f(r). Note that each of these expressions contains just one occurrence of R. In each of these expressions, we push the selection as far inside as possible. Selection distributes across union, commutes with another selection and can be pushed ahead of a projection. However, it distributes across a Cartesian product Y × Z only if the selection applies to components from just one of the two arguments Y and Z. The algorithm fails to commute the selection with the LFP operator if the (single) occurrence of r is in one of the arguments of a Cartesian product across which we cannot distribute the selection. We stop when this happens or when we find an expression of the form $h(g(\sigma_F(r)))$ and one of the previous expressions in the series is of the form $h(\sigma_F(r))$. In the latter case, the equivalent expression that we are looking for is h(LFP(s=g(s))), and we have succeeded in pushing the selection ahead of the LFP operator.

We note in conclusion that the expression f(r) must contain no more than one occurrence of *r*. For instance, the algorithm does not apply in this case:

$$\sigma_{a_1=john}(\mathrm{LFP}(a = a.a \cup p))$$

Aho and Ullman also present a similar strategy for commuting projections with the LFP operator, but we do not discuss it here.

5.2. Static Filtering

The Static Filtering algorithm is an extension of the Aho-Ullman algorithm described above. However, rules are represented as rule/goal graphs rather than as relational algebra expressions, and the strategy is described in terms of *filters* which are applied to the arcs of the graph. It is convenient to think of the data as flowing through the graph along the arcs. A *filter* on an arc is a selection which can be applied to the tuples flowing through that arc, and is used to reduce the number of tuples that are generated. Transforming a given rule/goal graph into an equivalent graph with (additional) filters on some arcs is equivalent to rewriting the corresponding set of rules.

The execution of a query starts with the nodes corresponding to the base relations sending all their tuples through all arcs that leave them. Each axiom node that receives tuples generates tuples for its head predicate and passes them on through all its outgoing arcs. A relation node saves all new tuples that it receives and passes them on through its outgoing arcs. Computation stops (with the answer being the set of tuples in the query node) when there is no more change in the tuples stored at the various nodes at some iteration. We note that this is simply Semi-Naive evaluation.

Given filters on all the arcs leaving a node, we can 'push' them through the node as follows. If the node is a relation node, we simply place the disjunction of the filters on each incoming arc. If the node is an axiom node, we place on each incoming arc the strongest consequence of the disjunction that can be expressed purely in terms of the variables of the literal corresponding to this arc.

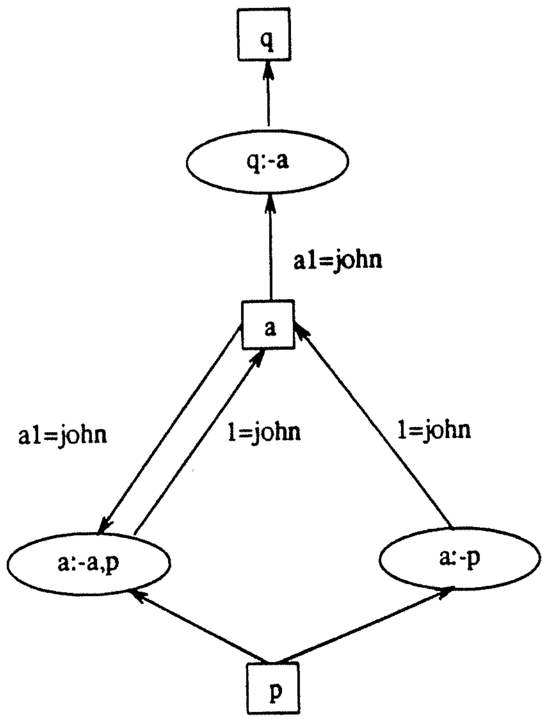
The objective of the optimization algorithm is to place the "strongest" possible filters on each arc. Starting with the filter which represents the constant in the query, it repeatedly pushes filters through the nodes at which the corresponding arcs are incident. Since the number of possible filters is finite, this algorithm terminates. It stops when further pushing of filters does not change the graph, and the graph at this point is equivalent to the original graph (although the graph at intermediate steps may not). Note that since the disjunction of 'true' with any predicate is 'true', if any arc in a loop is assigned the filter 'true', all arcs in the loop are subsequently assigned the filter 'true'.

Consider the transitive closure example that we optimized using the Aho-Ullman algorithm. We would represent it by the following axioms:

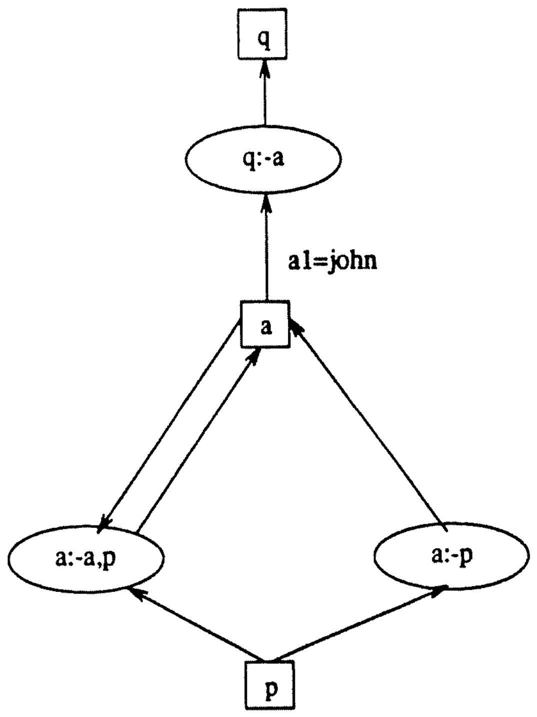
R1 a(X,Y) :- a(X,Z), p(Z,Y).
R2 a(X,Y) :- p(X,Y).
R3 q(X) :- a(john,X).

Given below is the corresponding system graph, before and after optimization (We have omitted the variables in the axioms for clarity):

After:



Before:



We begin the optimization by pushing the selection through the relation node a. Thus the arcs from R1 to *a* and from R2 to *a* both get the filter '1=john' (We have simplified the conventions for keeping track of variables - '1' refers to the first attribute of the corresponding head predicate). We then push these filters through the corresponding axiom nodes, R1 and R2. Pushing '1=john' through node R2 puts the filter 'p_1=john' on the arc from *p* to R2. Pushing '1=john' through node R1 puts the filter 'a_1=john' on the arc from *a* to R1. Note that it does not put anything on the arc from *p* to R1 (empty filters are equivalent to 'true'). There are no arcs entering *p*, and the filter on the arc from *a* to R1 does not change the disjunction of the filters on arcs leaving *a* (which is still 'a_1=john'). So the algorithm terminates here.

The analogy with the Aho-Ullman algorithm is easily seen when we recognize that a filter is a selection, pushing through a relation node is distribution across a $\cup$ and pushing through an axiom node is distribution across a Cartesian product. In general, the optimizations achieved by the two algorithms are identical. However, the Static Filtering algorithm is more general in that it successfully optimizes some expressions containing more than one occurrence of the defined predicate. An example is the expression

$$\sigma_{a_1=john}(LFP(a = (a.p \cup a.q \cup p)))$$

The Aho-Ullman algorithm does not apply in this case because there are two occurrences of R in f(R). The Static Filtering algorithm optimizes this to

$$LFP((\sigma_{a_1=john}(a) . p) \cup (\sigma_{a_1=john}(a) . q) \cup (\sigma_{a_1=john}(p)))$$

Essentially, it improves upon the Aho-Ullman algorithm in that it is able to distribute selection across some unions where both arguments contain r.

Further, the algorithm can work directly upon certain mutually recursive rules, for example

R1 r(X,Y) :- b(X), s(X,Y).
R2 s(X,Y) :- c(X), r(X,Y).
R3 q(X) :- r(X,john).

Before applying the Aho-Ullman algorithm, these rules must be rewritten as follows

R1 r(X,Y) :- b(X), c(X), r(X,Y).
R2 q(X) :- r(X,john).

Note that the Static Filtering algorithm fails to optimize both

$\sigma_{a_1=john}(LFP(a = a.a \cup p))$, and

$\sigma_{a_1=john}(LFP(a = a.p \cup p.a \cup p))$

In this description, we have treated only the "static" filtering approach of Kifer and Lozinskii. Elsewhere, they have also proposed "dynamic" filters (Kifer and Lozinskii [1986b]), which are not determined at compile time but are computed at run time, and this approach is similar to Generalized Magic Sets, discussed later in this section.

5.3. Magic Sets

The idea of the Magic Sets optimization is to simulate the sideways passing of bindings a la Prolog by the introduction of new rules. This cuts down on the number of potentially relevant facts.

The application domain is the set of bottom-up evaluable rules.

We shall describe the strategy in detail, using as an example a modified version of the same-generation rule set:

sg(X,Y) :- p(X,XP),p(Y,YP),sg(YP,XP).
sg(X,X).
query(X) :- sg(a,X).

Note that in this version the two variables XP and YP have been permuted. Note also that the second rule is not range restricted. The first step of the magic set transformation is the introduction of adornments and the generation of adorned rules.

Given a system of rules, the *adorned rule system* (Ullman [1985]) is obtained as follows:
For each rule r and for each adornment *a* of the predicate on the left, generate an adorned rule: Define recursively an argument of a predicate in the rule r to be *distinguished* (Henschen and Naqvi [1984]) if either it is bound in the adornment *a*, or it is a constant, or it appears in a base predicate occurrence that has a distinguished variable. Thus, the sources of bindings are (i) the constants and (ii) the bindings in the head of the rule. These bindings are propagated through the base predicates. If we consider each distinguished argument to be bound, this defines an adornment for each derived literal on the right. The adorned rule is obtained by replacing each derived literal by its adorned version.

If we consider the rule

sg(X,Y) :- p(X,XP),p(Y,YP),sg(YP,XP).

with adornment bf on the head predicate, then X is distinguished because it is bound in sg(X,Y), XP is distinguished because X is distinguished and p(X,XP) is a base predicate; these are the only distinguished variables. Thus the new adorned rule is

sg^{bf}(X,Y) :- p(X,XP),p(Y,YP),sg^{fb}(YP,XP).

If we consider a set of rules, this process generates a set of adorned rules. The set of adorned rules has size K.R where R is the size of the original set of rules and K is a factor exponential in the number of attributes per derived predicate. So, for instance, if every predicate has three attributes, then the adorned system is eight times larger than the original system. However, we do not need the entire adorned system and we only keep the adorned rules that derive the query. In our example the reachable adorned system is:

> sg^{bf} (X,Y) :- p(X,XP),p(Y,YP),sg^{fb} (YP,XP).
> sg^{fb} (X,Y) :- p(X,XP),p(Y,YP),sg^{bf} (YP,XP).
> sg^{bf} (X,X).
> sg^{fb} (X,X).
> $query^{f}$ (X) :- sg^{bf} (a,X).

Clearly, this new set of rules is equivalent to the original set in the sense that it will generate the same answer to the query.

The magic set optimization consists in generating from the given set of rules a new set of rules, which are equivalent to the original set with respect to the query, and such that their bottom-up evaluation is more efficient. This transformation is done as follows: (i) for each occurrence of a derived predicate on the right of an adorned rule, we generate a magic rule. (ii) For each adorned rule we generate a modified rule.

Here is how we generate the magic rule: (i) choose an adorned literal predicate p on the right of the adorned rule r, (ii) erase all the other derived literals on the right, (iii) in the derived predicate occurrence replace the name of the predicate by magic.p^{a} where *a* is the literal adornment, and erase the non distinguished variables, (iv) erase all the non distinguished base predicates, (v) in the left hand side, erase all the non distinguished variables and replace the name of the predicate by *magic.p* $1^{a'}$, where p1 is the predicate on the left, and a' is the adornment of the predicate p1, and finally (vi) exchange the two magic predicates.

For instance the adorned rule:

> sg^{bf} (X,Y) :- p(X,XP),p(Y,YP),sg^{fb} (YP,XP).

generates the magic rule:

> $magic^{fb}$ (XP) :- p(X,XP), $magic^{bf}$ (X).

Note that the magic rules simulate the passing of bound arguments through backward chaining. (We have dropped the suffix "sg" in naming the magic predicates since it is clear from the context.)

Here is how we generate the modified rule: For each rule whose head is p.a, add on the right hand side the predicate magic.p.a(X) where X is the list of distinguished variables in that occurrence of p. For instance the adorned rule:

> sg^{bf} (X,Y) :- p(X,XP),p(Y,YP),sg^{fb} (YP,XP).

generates the modified rule:

> sg^{bf} (X,Y) :- p(X,XP),p(Y,YP),$magic^{bf}$ (X), sg^{fb} (YP,XP).

Finally the complete modified set of rules for our example is:

> $magic^{fb}$ (XP) :- p(X,XP), $magic^{bf}$ (X).
> $magic^{bf}$ (YP) :- p(Y,YP),$magic^{fb}$ (Y).
> $magic^{bf}$ (a).
> sg^{bf} (X,Y) :- p(X,XP),p(Y,YP),$magic^{bf}$ (X),sg^{fb} (YP,XP).
> sg^{fb} (X,Y) :- p(X,XP),p(Y,YP),$magic^{fb}$ (Y),sg^{bf} (YP,XP).
> sg^{bf} (X,X) :- $magic^{bf}$ (X).
> sg^{fb} (X,X) :- $magic^{bf}$ (X).
> query.f(X) :- sg^{bf} (a,X).

The idea of the magic set strategy was presented in Bancilhon et al [1986] and the precise algorithm is described in Bançilhon et al [1986a]. A generalized version (Generalized Magic Sets, see below) has been implemented at MCC.

5.4. Counting and Reverse Counting.

Counting and Reverse Counting are derived from the magic set optimization strategy.

They apply under two conditions: (i) the data is acyclic and (ii) there is at most one recursive rule for each predicate, and it is linear.

We first describe counting using the "typical" single linear rule system:

```
p(X,Y) :- flat(X,Y).
p(X,Y) :- up(X,XU),p(XU,YU),down(YU,Y).
query(Y) :- p(a,Y).
```

The idea consists in introducing magic sets (called *counting* sets) in which elements are numbered by their distance to the element a. Remember that the magic set essentially marks all the *up* ancestors of a and then applies the rules in a bottom-up fashion to only the marked ancestors. In the counting strategy, at the same time we mark the ancestors of john, we number them by their distance from a. Then we can "augment" the p predicate by numbering its tuples and generate them by levels as follows:

```
counting(a,0).
counting(X,I) :-counting(Y,J),up(Y,X),I=J+1.
p'(X,Y,I) :- counting(X,I),flat(X,Y).
p'(X,Y,I) :- counting(X,I),up(X,XU), p'(XU,YU,J),down(YU,Y),I=J-1.
query(X) :- p'(a,X,0).
```

Thus at each step, instead of using the entire magic set, we only use the tuples of the correct level, thus minimizing the set of relevant tuples. But in fact, it is useless to compute the first attribute of the p predicate. Thus the system can be further optimized into:

```
counting(a,0).
counting(X,I) :-counting(Y,J),up(Y,X),I=J+1.
p''(Y,I) :- counting(X,I),flat(X,Y).
p''(Y,I) :- p''(YU,J),down(YU,Y),I=J-1,J>0.
query(X) :- p''(Y,0).
```

It is interesting to notice that this new set of rules is in fact simulating a stack.

Reverse counting is another variation around the same idea. It works as follow: (i) first compute the magic set, then (ii) for each element b in the magic set number all its *down* descendants and its *up* descendants and add to the answer all the *down* descendants having the same number as a (because a is in the *up* descendants). This gives the following equivalent system:

```
magic(a).
magic(Y) :- magic(X),up(X,Y).
des.up(X,X,0) :- magic(X).
des.down(X',Y,0) :- magic(X'),flat(X',Y).
des.up(X',X,I) :- des.up(X',Y,J),up(X,Y),I=J+1.
des.down(X',X,I) :- des.down(X',Y,J),down(Y,X),I=J+1.
query(Y) :- des.up(X',a,Y),des.down(X',Y,I).
```

This can be slightly optimized by limiting ourselves to the b's that will join with *flat* and restricting the *down* des's to be in the magic set. This generates the following system:

```
magic(a).
magic(Y) :- magic(X),up(X,Y).
des.up(X,X,0) :- magic(X),flat(X,Y).
des.down(X',Y,0) :- magic(X'),flat(X',Y).
des.up(X',X,I) :- magic(X),des.up(X',Y,J),up(X,Y),I=J+1.
des.down(X',X,I) :- des.down(X',Y,J),down(Y,X),I=J+1.
sg(a,Y) :- des.up(X',a,Y),des.down(X',Y,I).
```

Note that we still have the problem of a "late termination" on *down* because we number *all* the descendants in *down*, even those of a lower generation than a.

The idea of counting was presented in Bancilhon et al [1986] and a formal description of counting and of an extension called "magic counting" was presented in the single rule case in Sacca and Zaniolo [1986a]. Counting was extended to progams containing function symbols in Sacca and Zaniolo [1986b]. Reverse counting is described in Bancilhon et al. [1986a]. A generalized version of Counting (Generalized Counting, Beeri and Ramakrishnan [1987]) has been implemented at MCC.

5.5. Generalized Magic Sets

This is a generalization of the Magic Sets method and is described in Beeri and Ramakrishnan [1987]. The intuition is that the Magic Sets method works essentially by passing bindings obtained by solving body predicates "sideways" in the rule to restrict the computation of other body predicates. The notion of *sideways information passing* is formalized in terms of labeled graphs. A sideways information passing graph is associated with each rule, and these graphs are used to define the Magic Sets transformation. (In general, many such graphs exist for each rule, each reflecting one way of solving the predicates in the body of the rule; and we may choose any one of these and associate it with the rule.)

There are examples, such as transitive closure defined using double recursion, in which the original Magic Sets transformation achieves no improvement over Semi-Naive evaluation. Intuitively, this is because the only form of sideways information passing that it implements consists of using base predicates to bind variables. Thus, in the same generation example discussed earlier, the predicate *p* is used to bind the variable XP. The method, however, fails to pass information through derived predicates, and so it fails with transitive closure expressed using double recursion (since the recursive rule contains no base predicates in the body). Consider the rule:

a(X,Y) :- a(X,Z), a(Z,Y).

Given a query a(john,Y), the Magic Sets method recognizes that X is bound (since it is bound in the adornment *bf* corresponding to the head of the rule). However, Z is considered free. So it generates the following adorned rule:

a^{bf}(X,Y) :- a^{bf}(X,Z), a^{ff}(Z,Y).

Clearly, the method computes the entire ancestor relation. To succeed in binding Z, the first occurrence of *a* in the body must be used.

The generalized version of the method succeeds in passing information through derived predicates as well.

As with the original Magic Sets strategy, a set of *adorned rules* is first obtained from the given rules, and these adorned rules are then used to produce the optimized set of rules. Both these steps are now directed, however, by the notion of *sideways information passing graphs* (sips). A sip corresponding to the above rule that binds Z is:

$h \rightarrow_X a.1,\ h, a.1 \rightarrow_Z a.2$

The predicate *h* denotes the bound part of the head. This graph indicates that the head binds X and this is used in solving the first occurrence of *a*, and further, this solution is used to bind Z in solving the second occurrence of *a*. This generates the adorned rule:

a^{bf}(X,Y) :- a^{bf}(X,Z), a^{bf}(Z,Y).

The magic rules corresponding to the two occurrences of *a* are:

magic(X) :- magic(X).
magic(Z) :- magic(X), a^{bf}(X,Z).

The first rule is trivial and may be discarded. In addition, we obtain the rule magic(john) from the query. The modified rules are obtained exactly as in the Magic Sets method, by adding magic predicates to the bodies of the original rules.

We do not present the details here. The reader is referred to Beeri and Ramakrishnan [1987], where Counting and variants of both Magic Sets and Counting are generalized as well. Generalized Magic Sets and Generalized Counting have been implemented at MCC.

The work in Beeri and Ramakrishnan [1987] still imposes one restriction on rules: every variable that appears in the head of a rule must also appear in the body. This ensures that every tuple produced in a bottom-up execution is ground. † On the other hand, this restriction disallows the use of certain effective logic programming techniques, such as difference lists, and makes it difficult to utilize partially bound arguments. This restriction is lifted in Ramakrishnan [1988], and thus, it is shown that the rewriting techniques (i.e., Magic Sets, Counting, etc.) can be generalized to deal with arbitrary logic programs.

The "Alexander" strategy described in Rohmer et al. [1986] is essentially a variant of the Generalized Magic Sets strategy.

The dynamic filtering approach of Kifer and Lozinskii is similar to the Generalized Magic Sets strategy, although it cannot implement some sideways information passing graphs. The dynamic filters essentially perform as magic sets, but this is a run-time strategy, and the overhead of computing and applying the filters falls outside our framework. We do not discuss dynamic filtering further in this paper.

6. Summary of Strategy Characteristics.

A summary of the characteristics of each strategy is presented in Table 1. We emphasize that this table contains some approximations and refer the reader to the actual descriptions for clarifications. For example, although Magic Sets and Generalized Magic Sets have the same application range, the latter succeeds in optimizing some queries in this range that the former cannot. (Magic Sets essentially reduces to Semi-Naive evaluation in these cases.) Several of these strategies have been further developed since this paper was written. We refer the reader to Beeri and Ramakrishnan [1987], Ramakrishnan [1988], Kifer and Lozinskii [1986b, 1988], Sacca and Zaniolo [1987] and Vieille [1988] for some of these further developments. (We also note that there has been significant related research that cannot be viewed as development of work discussed in this paper. It is outside the scope of this paper to review this work, and the interested reader is urged to consult the recent database literature. The collection of papers in Minker [1988] is a good starting point.)

Table 1: Summary of Strategy Characteristics

Method	*Application Range*	*Top down vs. Bottom Up*	*Compiled vs. Interpreted*	*Iterative vs. Recursive*
Naive Evaluation	Bottom-up Evaluable	Bottom Up	Compiled	Iterative
Semi-Naive Evaluation	Bottom-up Evaluable	Bottom Up	Compiled	Iterative
Query/Subquery	Range Restricted No Arithmetic	Top Down	Interpreted	Iterative
Query/Subquery	Range Restricted No Arithmetic	Top Down	Interpreted	Recursive
APEX	Range Restricted No Arithmetic Constant Free	Mixed	Mixed	Recursive
Prolog	User responsible	Top Down	Interpreted	Recursive

† The reader who is familiar with logic programs should note that this restriction makes an important optimization possible - the expensive operation of *unification* can always be replaced by the less expensive operation of *matching*, since one of the two arguments to the unification procedure is always ground, given that all generated tuples are ground.

Henschen-Naqvi	Linear	Top Down	Compiled	Iterative
Aho-Ullman	Strongly Linear	Bottom Up	Compiled	Iterative
Kifer-Lozinskii	Range Restricted No Arithmetic	Bottom Up	Compiled	Iterative
Counting	Strongly Linear	Bottom Up	Compiled	Iterative
Magic Sets	Bottom-up evaluable	Bottom Up	Compiled	Iterative
Generalized M. Sets	Bottom-up evaluable	Bottom-up	Compiled	Iterative

7. Framework for Performance Evaluation

We now turn to the problem of comparing the above strategies. To perform a comparison of the strategies we must:

1. Choose a set of rules and queries which will represent our benchmark.
2. Choose some test data which will represent our extensional database.
3. Choose a cost function to measure the performance of each strategy.
4. Evaluate the performance of each query against the extensional databases.

We first describe the four queries used as "typical" intensional databases. Then, we present our characterization of the data. Each relation is characterized by four parameters and it is argued that a number of familiar data structures, e.g. trees, can be described in this framework. We describe our cost metric, which is the size of the intermediate results before duplicate elimination. We present analytical cost functions for each query evaluation strategy on each query. The cost functions are plotted for three sets of data - tree, inverted tree and cylinder. We discuss these results informally.

7.1. Workload: Sample Intensional Databases and Queries

Instead of generating a general mix, we have chosen four queries that have the properties of exercising various important features of the strategies. We are fully aware of the fact that this set is insufficient to provide a complete benchmark, but we view this work as a first step towards a better understanding of the performance behavior of the various strategies.

The queries are three different versions of the ancestor query and a version of the same-generation query. The first one is just a classical ancestor rule and query with the first attribute bound.

Query 1
```
a(X,Y) :- p(X,Y).
a(X,Y) :- p(X,Z),a(Z,Y).
query(X) :- a(john,X).
```

Because most strategies are representation dependent, we have studied the same example with the second attribute bound instead of the first. This will allow us to determine which strategies can solve both cases.

Query 2
```
a(X,Y) :- p(X,Y).
a(X,Y) :- p(X,Z),a(Z,Y).
query(X) :- a(X,john).
```

The third version of the ancestor example specifies ancestor using double recursion. This enables us to see how the strategies react to the non linear case. This example being fully symmetric, it is sufficient to test it with its first attribute bound.

Query 3
```
a(X,Y) :- p(X,Y).
a(X,Y) :- a(X,Z),a(Z,Y).
```

```
query(X) :- a(john,X).
```

Finally, to study something more complex than transitive closure, we have chosen a generalized version of the same generation example, bound on its first attribute.

Query 4
```
p(X,Y) :- flat(X,Y).
p(X,Y) :- up(X,XU),p(XU,YU), down(YU,Y).
query(X) :- p(john,X).
```

7.2. Characterizing Data: Sample Extensional Databases

Because we decided on an analytical approach, we had to obtain tractable formulae for the cost of each strategy against each query. Therefore, each relation must be characterized by a *small* set of parameters. Fortunately, because of the choice of our workload, we can restrict our attention to binary relations.

We represent every binary relation by a directed graph and view tuples as edges and domain elements as nodes. Nodes are arranged in layers and each edge goes from a node in one layer to a node in the next. Note that in these graphs each node has at least one in-edge or one out-edge. Nodes in the first layer have no incoming edges and nodes in the last layer have no outgoing edges. We assume that edges are randomly distributed with a uniform distribution.

This formalism does not represent cycles. Nor does it represent short cuts, where a short cut is the existence of two paths of different length going from one point to another. Clearly, they would violate our assumption that nodes were arranged in layers with edges going from nodes in one layer to the next.

Let R be a binary relation and A be a set. Recall that we denote by A.R the set:

$$A.R = \{ y \mid x \in A \text{ and } R(x,y) \}$$

We characterize a binary relation R by:

(1) F_R the *fan-out* factor,
(2) D_R the *duplication* factor,
(3) h_R the *height*, and
(4) b_R the *base*.

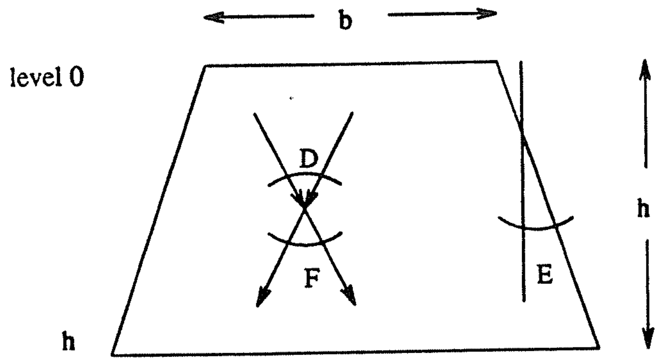
F_R and D_R are defined as follows: given a "random" set A of n nodes from R, the size of A.R is n F_R before duplicate elimination. D_R is the duplication factor in A.R, i.e. the ratio of the size of A.R before and after duplicate elimination. Thus the size of A.R after duplicate elimination is n F_R/D_R.

We call $E_R = F_R/D_R$ the *expansion* factor of R.

The base b_R is the number of nodes that do not have any antecedents. The height h_R is the length of the longest chain in R.

When no confusion is possible, we shall simply use F, D, h and b instead of F_R, D_R, h_R and b_R.

The typical structure consists of a number of layers. There are (h_R+1) layers of nodes in the structure, numbered from top to bottom (as 0 to h). There are b_R nodes in level 0.



This "parametrized structure" is fairly general and can represent a number of typical configurations:

A binary balanced tree of height k is defined by:
F=2; D=1; h=k; b=1
The same binary tree upside down is defined by:
F=1; D=2; h=k, $b=2^k$
A list of length k is defined by:
F=1; D=1; h=k; b=1
A set of n lists of length k is defined by:
F=1; D=1; h=k; b=n
A parent relation, where each person has two children and each child has two parents is defined by:
F=2; D=2; h=number of generations; b=number of people of unknown parentage

We emphasize that we assume the data to be *random*, with a uniform distribution. Thus, the values F and D are average values. Our characterization of a binary tree, for instance, describes a random (but layered) data structure in which the average values of F and D are 2 and 1 respectively. An actual binary tree has a regular pattern (*each* internal node has exactly one incoming and two outgoing edges incident on it) and this is not captured by our characterization.

Our assumption that the duplication factor is independent of the size is a very crude approximation. For instance it implies that if you start from one node you still generate some duplicates. Obviously the duplication factor increases with the size of the start set. Therefore, our approximation overestimates the number of duplicates. However, it becomes reasonable as the size of the start set becomes large. It is also dependent upon our assumption that the data is random and not regular.

Let us now turn to the problem of characterizing inter-relation relationships. Let A and B be two sets. The *transfer ratio* of A with respect to B, denoted $T_{A,B}$ is the number such that given a random set of n nodes in A, the size of $A \cap B$ after duplicate elimination is n $T_{A,B}$. In other words, given a set of nodes in A, the transfer ratio is the fraction of these nodes that also appear in set B. Note that $0 \le T \le 1$.

This definition can be extended to binary relations by considering only the columns of the relations. We shall denote the i-th column of R by Ri. Thus, given two binary relations R and S, the number of tuples in the (ternary) result of the join of R and S is n $T_{R2,S1}$, where n is the number of tuples in R.

7.3. The Cost Metrics

We have chosen for our cost measure the number of *successful inferences* performed by the system.

Consider a rule:

p :- $q1, q2, \ldots, qn$

A *successful inference* (or *firing*) associated with this rule is of the form (*id, t, t1, t2, ... , tn*), where *t1* through *tn* are (ground) tuples in *q1* through *qn* and *t* is a tuple in *p*. It denotes that the truth of *t1* through *tn* is used to establish that *t* is true, by applying the given rule. There is (conceptually) an identifier *id* associated with this inference because it is possible that this inference is repeatedly made, and we wish to measure this.

The simplest way to obtain this cost function is to measure the size of the intermediate results *before* duplicate elimination.

We note that this cost measure does not count *unsuccessful* inferences, i.e. uses of the rule in which the tuples *t1* through *tn* fail to establish *t* (for example, because they do not agree on the values they assign to common variables). Also, since the cost measure is independent of the number of *q*'s, in this model the measure of complexity of the join, the cartesian product, intersection and selection is the size of the result; the measure of complexity of union is the sum of the sizes of the arguments (each tuple present in both argument is going to fire twice); and the measure of complexity of projection is the size of the argument. Readers familiar with performance evaluation of relational queries might be surprised by these measures.

Our concern, however, is primarily with recursive queries. In particular, all but one of our queries (ancestor using double recursion) are *linear*, i.e., the body of each recursive rule contains exactly one occurrence of the recursive predicate. We justify our measurement of only successful inferences by the observation that the number of successful inferences (for the recursive predicate) at one step constitutes the operand at the next step. We justify the approximation in estimating the cost of a join in terms of the size of just one of the operands as follows. The join represented by the predicates in the body of a rule may be thought of as a fixed "operator" that is repeatedly applied to the relation corresponding to the recursive predicate. It is reasonable to assume that the cost of each such application is proportional to the size of this relation (the operand). By measuring the size of this intermediate relation over all steps, we obtain a cost that is proportional to the actual cost.

In essence, our cost is a measure of one important factor in the performance of a query evaluation system, the number of successful inferences, rather than a measure of the actual run-time performance. This cost model is studied further in Bancilhon [1985].

8. Notation and Preliminary Derivations

In this section, we explain the notation and terminology used in analytically deriving the cost functions. We also derive some expressions that are used in the analysis of some of the strategies. The derivations of these expressions are of some interest in their own right, since they are good examples of the techniques we use in subsequent analyses.

We denote multiplication by simply juxtaposing the operands. Where there is ambiguity, parentheses are used to clarify the expression, or we use * to denote multiplication.

We denote the number of nodes at level i in relation R by $n_R(i)$, the total number of nodes in R by N_R, and the total number of edges in R (which is the number of tuples in R) by A_R. Where no confusion is possible, we drop the subscripts.

We denote the sum of the (h+1)st elements of the geometric series of ratio E by gsum(E,h), thus:

$$gsum(E,h) = (1 + E + E^2 + E^3 + ... + E^h)$$

From the definition of the expansion factor E, we have n(i+1)=n(i)E. So the total number of nodes is:

$$\begin{aligned} N &= b(1 + E + E^2 + E^3 + ... + E^h) \\ &= (b)gsum(E,h) \end{aligned}$$

Clearly, the number of edges entering level i is n(i-1)F, and the number of edges leaving level i is n(i)F. Thus the total number of edges is:

$$\begin{aligned} A &= bF + bEF + bE^2F + ... + bE^{(h-1)}F \\ &= bF(1 + E + E^2 + ... + E^{(h-1)}) \\ &= (bF)gsum(E,h-1) \end{aligned}$$

We denote by h' the *average* level:

$$h' = h - \left\lfloor \frac{\sum_{i=1}^{h}(i*n(i))}{N} \right\rfloor$$

It denotes the mean level at which we pick a node, assuming nodes are uniformly distributed. We have actually defined h' as the distance of the mean level from the highest level h for notational convenience, since this is a quantity we use extensively.

We define the length of an arc (a,b) in the transitive closure of R (which we denote by R*) to be the length of the path of R that generates it. (Note that this is well-defined because there are no short-cuts.)

Since an arc is represented by its end points, the number of arcs of length k with a given first node can be computed as the number of distinct nodes reachable from the given node by a path of length k. So, starting from a given node, on the average we can reach E distinct nodes by a path of length 1, E^2 distinct nodes by a path of length 2, and so on. The number of arcs of length k going from level i to level (i+k) is thus:

$$n(i)E^k = n(i+k)$$

Of course, if D is not one, this is an approximation that depends on our assumption of random data. In particular, it breaks down for regular data, such as an actual inverted tree. The intuition is as follows. The parameters F and D are used to estimate the number of *arcs* of length k, as opposed to the number of *paths* of length k. Several paths may generate the same arc (i.e., they have the same end points). Thus, we use the parameters F and D to estimate this "duplication" of arcs. This approximation depends upon the randomness of data - in an inverted tree, for instance, the number of paths is exactly the number of arcs because there is a unique path between any two points. The inverted tree is one instance of a family of data structures with given values of F (=1) and D (=2), and in this particular instance, due to the regular pattern in the data, the above approximation breaks down. In general, however, for such a structure the number of paths is *not* equal to the number of arcs; and if the data is randomly (and uniformly) distributed, our approximation is accurate.

We denote by $a_{R^*}(k)$ the number of arcs of length exactly k in R*. Where the context is clear, we write a(k).

a(k) is obtained by summing all the arcs of length k that enter level i for i = k to h. Thus:

$$\begin{aligned} a(k) &= n(k) + n(k+1) + \dots + n(h) \\ &= n(k)\, \mathrm{gsum}(E, h-k) \end{aligned}$$

Finally, given a relation R(A,B), its transpose $R^T(B,A)$ is defined to be such that $R^T(B,A)$ holds iff R(A,B) holds, for all pairs (A,B). We have the following relationships:

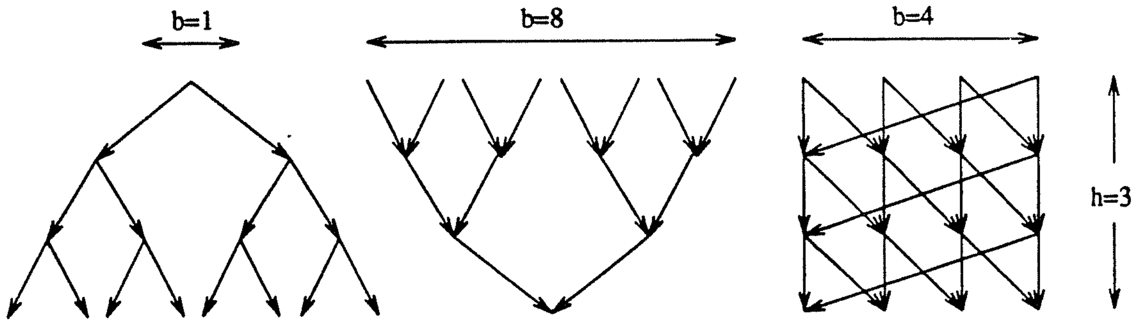
$F_{R^T} = D_R$,
$D_{R^T} = F_R$,
$E_{R^T} = 1/E_R$,
$h_{R^T} = h_R$,
$h'_{R^T} = h_R - h'_R$, and
$b_{R^T} = b_R E_R^{h_R}$.

9. Cost Evaluation

For each strategy and for each query, we have analytically evaluated the cost of computing the given query using the given strategy. The cost is expressed as a function of the data parameters F, D, h and b. The formulae are listed in Appendix 1, and their derivations are contained in Bancilhon and Ramakrishnan [1988]. To compare these fairly complex formulae, we have plotted a number of curves, some of which are included in the appendix.

10. Graphical Comparison of the Costs

The curves shown in the appendix show the relative performance of the various strategies on each of the sample queries for three sets of data. They are relations in which the tuples are arranged in a tree structure, an inverted tree structure, and a "cylinder". A cylinder is a structure in which each layer has b nodes and each node has on the average two incoming and two outgoing arcs. We present below a sample relation of each type:



Tree, F=2, D=1 Inverted tree, F=1, D=2 Cylinder, F=2, D=2

The choice of these structures was made in order to study the effects of uneven distribution of the data and the effects of duplication. We have fixed the sizes of all relations at 100,000 tuples. For the tree structure, we vary the shape by changing the fan-out F while keeping the number of arcs (which is the number of tuples) constant. Clearly, decreasing the fan-out increases the depth of the structure and vice-versa. Similarly, the shape of the inverted tree is varied by varying the duplication factor. The shape of the cylinder is varied by varying the ratio of breadth b to height h, again keeping the number of arcs constant.

For each query and data structure, we plot the cost of each strategy against the shape of the data (measured in terms of the parameter used to vary it). Thus, for each query, we plot cost vs. F for the tree, cost vs. D for the inverted tree, and cost vs. b/h for the cylinder. We do this for each strategy. The cost is computed using the cost functions listed in the appendix. We have sometimes displayed a subset of the curves (for the same query and data structure) over a different range, to allow a better comparison.

For the ancestor queries, we plot the cost of each strategy for the cases when the parent relation has 100,000 tuples and the data in it has the shape of a tree, an inverted tree and a cylinder.

For the same generation example, we have assumed that the relations *up* and *down* are identical and that the fan-out and duplication for the relation flat are both equal to 1. We have also assumed that the transfer ratio from up to flat is equal to the transfer ratio from flat to down. We have assumed that all three relations (*up*, *flat* and *down*) have 100,000 tuples. We plot the cost of each strategy as the shape of *up* and *down* varies for a total of six cases: the cases when the structure is a tree, an inverted tree and a cylinder, with the transfer ratio equal to 1 and 0.01 (100% and 1% respectively).

10.1. Summary of the Curves

There are several important points to be seen in the curves. For a given query, there is a clear ordering of the various strategies that usually holds over the entire range of data. The difference in performance between strategies is by orders of magnitude, which emphasizes the importance of choosing the right strategy. The cost of the optimal strategy is less than 10,000 in each of the queries we have considered, over the entire range of data. The size of the data is 100,000 tuples. This indicates that recursive queries can be implemented efficiently.

We present a summary of the ordering of the strategies, as seen in the corresponding curves. We use $\ll$ to denote an order of magnitude or greater difference in performance, and for a given query, we list in parentheses those strategies that perform identically for all data. We refer to the various strategies using the following acronyms for brevity: HN (Henschen-Naqvi), C (Counting), MS (Magic Sets), GMS (Generalized Magic Sets), QSQR, QSQI, APEX, P (Prolog), SN (Semi-Naive), N (naive) and SF (Static Filtering).

Query 1 (Ancestor.bf)

Tree: (HN,C) $\ll$ (QSQR,APEX) = P $\approx$ (MS,GMS) $\ll$ QSQI $\ll$ (SN,SF) $\ll$ N

Inverted tree: (HN,C) $\ll$ (QSQR,APEX) $\approx$ (MS,GMS) $\ll$ P $\ll$ QSQI $\ll$ (SN,SF) $\ll$ N

Cylinder: (HN,C) $\ll$ (QSQR,APEX) $\approx$ (MS,GMS) $\ll$ QSQI $\ll$ (SN,SF) $\ll$ N $\ll$ P

Query 2 (Ancestor.fb)

All Data: (HN,MS,GMS,QSQR,SF) $\ll$ APEX $\approx$ QSQI $\ll$ SN $\ll$ N $\approx$ P

Query 3 (Ancestor.bf, non-linear)

All data: QSQR $\approx$ GMS $\ll$ QSQI $\ll$ APEX $\ll$ (SN,MS,SF) $\ll$ N

(HN, Counting and Prolog do not apply)

Query 4 (Same Generation.bf)

Tree: C $\ll$ HN $\approx$ (MS,GMS) $\ll$ QSQR = P $\ll$ APEX $\ll$ QSQI $\ll$ (SN,SF) $\ll$ N

Inverted tree: C $\ll$ HN $\approx$ (MS,GMS) $\ll$ QSQR $\approx$ APEX $\ll$ P $\approx$ QSQI $\ll$ (SN,SF) $\ll$ N

Cylinder: C $\ll$ HN $\approx$ (MS,GMS) $\ll$ QSQR $\ll$ APEX $\ll$ QSQI $\ll$ (SN,SF) $\ll$ N $\ll$ P

To summarize the ancestor results, the following order is seen to hold for the ancestor queries:

(HN, C) $\ll$ QSQR $\approx$ (MS,GMS) $\ll$ APEX $\approx$ QSQI $\ll$ SN $\ll$ N

There are some exceptions and additions to the above ordering. In the non-linear case, Henschen-Naqvi and Counting do not apply, and Magic Sets reduces to Semi-Naive. Static Filtering performs like Semi-Naive, except in the case where the second argument is bound, and in this case it performs like QSQR. APEX performs like QSQR in the case where the first argument is bound. Prolog performs poorly when it cannot propagate the constant in the query (the case where the second argument is bound), as expected. When it can propagate the constant, its performance degrades sharply with duplication, especially as the depth of the data structure increases. This is readily seen from the curves for the cylinder.

To summarize the same generation results, we have:

$C \ll HN \approx (MS,GMS) \ll QSQR \ll APEX \ll QSQI \ll (SN, SF) \ll (P, N)$

Prolog behaves like QSQR when there is no duplication (tree). With duplication, its performance degrades so sharply with an increase in the depth of the data structure that we have classified it with Naive, although it performs better than Semi-Naive over a wide range.

10.2. Interpreting the Results

These results indicate that the following three factors greatly influence the performance:

1. The amount of *duplication* of work,
2. The size of the set of *relevant facts*, and
3. The size (number and arity) of intermediate relations.

By duplication of work, we refer to the repeated firing of a rule on the same data. This can occur due to duplication in the data (e.g. Prolog), or due to an iterative control strategy that does not remember previous firings (e.g. QSQI and Naive). Consider the second factor. A fact p(a) is *relevant* to a given query q iff there exists a derivation $p(a) \rightarrow^* q(b)$ for some b in the answer set. If we know all the relevant facts in advance, instead of using the database to reply to the query, we can use the relevant part of the database only, thus cutting down on the set of facts to be processed. It is in general impossible to find all the relevant facts in advance without spending as much effort as in replying to the query. Thus, all methods have a way of finding a super-set of relevant facts. We call this set the *set of potentially relevant facts.* As this set becomes smaller, i.e., contains fewer and fewer facts that are *not* relevant, the work done in evaluating the query clearly decreases. The third factor is hard to define precisely. Strategies that only look at sets of nodes rather than sets of arcs perform better than those that look at sets of arcs, by an order of magnitude or more. They are less generally applicable since this often involves a loss of information. This usually leads to non-termination unless the database has certain properties, such as linearity of rules and acyclicity of the extensional database. And of course, strategies that create more intermediate relations pay for it in increased costs, since the addition of a tuple to a relation (intermediate or otherwise) represents a firing.

The following discussion is intended to clarify these concepts, as well as to explain the performance of the various strategies in terms of these three factors.

10.2.1. The Ancestor Queries

We begin by looking at the ancestor queries. The effect of duplication is seen by considering Prolog and QSQI, both of which do duplicate work, for different reasons. When the first argument is bound, Prolog performs like QSQR on a tree data structure, where exactly one arc enters each node (equivalently, there is exactly one way of deriving a given answer). With duplication (i.e. on the average more than one arc enters a given node) performance degrades dramatically. Prolog's performance for the same query on a cylinder is comparable to Naive evaluation, a difference of several orders of magnitude! We note that the set of relevant facts is comparable in the two cases, being the set of nodes reachable from the node denoting the constant in the query (which will henceforth be referred to as the query node). However, in the case of the cylinder, these nodes can be reached along several paths and Prolog infers them afresh along each path. QSQI performs duplicate computation for a different reason, which is that its iterative control strategy does not remember previous firings. Essentially, there are as many steps (executions of the control loop) as the longest path from the query node, and all nodes reached by a path of length less than or equal to i are recomputed at all steps after the ith. This can be seen by comparing QSQR and QSQI and noting that QSQI is orders of magnitude worse in all cases. QSQR uses the same set of relevant facts (the reachable nodes) and differs only in that it has a recursive control strategy that avoids precisely this duplication. Naive evaluation also does a lot of duplicate work, for the same reason as QSQI, i.e., it does not remember previous firings. Semi-Naive differs from Naive only in that it remembers all previous firings and does not repeat them. Thus, the effect of duplication can also be seen in the difference between Naive and Semi-Naive.

The effect of a smaller set of relevant facts can be seen in the vast difference between Magic Sets and Semi-Naive. Magic Sets is simply Semi-Naive applied to the set of relevant facts, which is determined to be the set of reachable nodes except in the doubly recursive case. In this case, the first phase of the Magic Sets strategy, which computes the set of relevant facts, fails and the Magic Sets strategy degenerates to Semi-Naive. This effect can also be seen in the behavior of Prolog on a tree data structure (which means we eliminate the effect of duplication) when the first argument is free. Prolog's depth first strategy is unable to propagate the constant in the second argument of the query. In other words, it must consider all facts in the database, and its performance degrades by several orders of magnitude. Similarly, the Static Filtering strategy degenerates to Semi-Naive when the optimization algorithm fails to push down the constant in the query. We note that pushing the constant (i.e., the selection that it represents) is equivalent to cutting down on the number of relevant facts.

QSQR succeeds in restricting the set of relevant facts to the set of nodes reachable from the query node even in the non-linear version of ancestor. It does this at the cost of implementing the recursive control, which is a cost that we do not understand at this stage. QSQI also succeeds in restricting the set of relevant facts, but performs a great deal of duplicate computation. The Magic Sets algorithm uses the entire parent relation for the set of relevant facts and so degenerates to Semi-Naive. APEX, for reasons explained below, also uses a much larger set of relevant facts. So, although it improves upon Semi-Naive computation in this case, it is much worse than QSQR. The Generalized Magic Sets strategy, however, succeeds in restricting the set of relevant facts to those reachable from the query node, thus illustrating its wider applicability. Henschen-Naqvi and Counting do not apply and Prolog does not terminate.

The behavior of APEX illustrates the interesting distinction between the set of relevant facts and the set of *useful* facts. The first step in the APEX strategy is to find what APEX calls the set of relevant facts (which is actually a subset of the set of relevant facts as we have defined it, since it does not include all facts than could derive an answer). In the ancestor examples, these are facts from the relation parent, and the firing of the first rule adds them to the ancestor relation. Subsequently, these facts are substituted (in turn) into both the parent and ancestor predicates in the body of the second rule. Except in the first case, this leads to subqueries whose answers are not relevant. For example, in the case where the second argument is bound to john, the set of relevant (a la APEX) facts is the set of facts p(X,john). By substituting these into the parent predicate in the second rule, we generate the query a(john,?). This computes the ancestors of john, whereas the given query a(?,john) asks for the descendants of john. This is because APEX does not make the distinction that facts of the form p(X,john) are relevant to the query a(?,john) only when substituted into the ancestor predicate in the second rule. This is a distinction that the Magic Sets strategy makes, and it thereby reduces the number of useless firings.

We now consider the third factor, the arity of the intermediate relations. The two strategies that use unary intermediate relations are the Henschen-Naqvi and Counting strategies. In essence, at step i they compute the set of relevant facts that is at a distance i from the query node. Let us denote this set by Si. At the next step, they compute the set of those nodes in parent to which there is an arc from a node in Si. Thus, they compute all nodes reachable from john, and further they compute each node at most D times where D is the duplication factor. However, the unary relations strategy fails to terminate if the query node is in a cycle. Also, neither the Henschen-Naqvi nor the Counting strategy applies when there are non-linear rules.

Magic Sets computes exactly the same set of relevant facts and does no duplicate work. However, in the second phase at step i it computes all arcs in the transitive closure of parent (restricted to the set of relevant facts) of length i. In particular, this includes all arcs of length i rooted at john. This is the answer, and this is essentially all that the more specialized methods, Henschen-Naqvi and Counting, compute. Everything else that the Magic Sets strategy does is useless computation. Thus, the cost of the Magic Sets strategy is the number of arcs in the transitive closure of the subtree rooted at john (i.e. the subtree of nodes reachable from john).

The recursive control of QSQR generates subqueries using precisely the nodes in set Si at step i, and the answer to each of these subqueries is the set of all nodes in the subtree rooted at that node. By induction, it is easy to see that the total cost involved in computing a query is the number of arcs in the transitive closure of the subgraph rooted at that query node. (The cost is thus similar to that of Magic Sets.) The intermediate relations here are the (binary) sets of answers to each subquery. This seems to indicate the power of a recursive control strategy since it succeeds in reducing both the set of relevant facts and the amount of duplicate work.

10.2.2. The Same Generation Query

We conclude this discussion by explaining the performance of the various strategies in the same generation query in terms of these three factors. Counting has the best performance since it uses the smallest set of relevant facts (the nodes of *up* that are reachable from the query node), does not do duplicate computation, and further, uses unary intermediate relations. It executes the query in two phases. In the first phase, at step i, it computes the set of all nodes in *up* that are reachable from the query node via a path of length i. In the second phase, it first computes the nodes of *down* that are reachable from this set via an arc of flat, still retaining the distance of each set from the query node. In subsequent iterations, it steps through *down* once each time, such that each node in a set that is i steps away from the query node in *up* is the root of paths of length i in *down*.

Henschen-Naqvi uses the same set of relevant facts, and is a unary strategy, but it does a lot of duplicate work. It is a single phase algorithm that does the same amount of work as the first phase of Counting in computing sets of *up* nodes along with their distances from the query node. However, it steps through *down* i times for each set at a distance i from the query node in *up*. Since it does not keep track of the work it does in step i at step i+1, it repeats a lot of the work in stepping through *down*. (Unless, of course, the data is such that there is no duplication of work. This corresponds to the data configuration in the worst case for Counting - the additional book-keeping done by Counting is unnecessary since the data ensures that there is no duplication of work in stepping through *down*.)

The set of relevant facts for Magic Set and QSQR is again the set of *up* nodes reachable from the query node. They do not perform duplicate computation. However, they work with binary relations, in effect computing all paths with equal lengths in *up* and *down* linked by a single arc in flat. Thus, their performance is inferior to that of Counting. Further, QSQR's left to right strategy forces us to create intermediate relations for up^* and $up^*.sg$, where up^* denotes the transitive closure of *up*. Since the Magic Set strategy does not impose any order of evaluation, we can do with the single intermediate relation *sg*. The cost of the additional inferences required to create the intermediate relations causes a large difference in the costs of the two strategies.

Our graphs show the performance of Magic Set to be identical to that of Henschen-Naqvi. It is to be expected that they perform similarly since the duplicate work done by Henschen-Naqvi is offset by the fact that they work with binary relations. However, their performance is not really identical. It appears to be so in our curves for two reasons. The first is our approximation of the number of arcs of length l to n(l)gsum(E,h-l). The second is the fact that we plot the curves for cases where *up* and *down* are identical. Under these conditions, the expressions for the performance of these methods become identical.

QSQI is similar to QSQR except that at each step, it duplicates the work of the previous steps, and so it is inferior to Magic Set and QSQR. Semi-Naive uses binary relations, and although it does not do duplicate work, this is outweighed by the fact that the set of relevant facts is all the nodes in *up*. So it performs worse than QSQI. Static Filtering degenerates to Semi-Naive since the optimization strategy fails to make any improvements to the system graph. Prolog is similar to QSQR when there is no duplication in the data, but its cost increases exponentially with the depth of the data structure when there is duplication. Naive evaluation uses the entire set of nodes in *up* as relevant facts, does duplicate work since it does not remember firings, and uses binary intermediate relations. With the exception of Prolog over a certain range, it is clearly the worst strategy.

Finally, we note that when the transfer ratio T is 0.01 (1%), the cost of computing the answer by Naive or Semi-Naive evaluation is essentially that of computing all arcs in the relation *flat*, and so the two methods perform almost identically.

11. Related Work

The performance issue was addressed informally through the discussion of a set of examples in Bancilhon et al. [1986b]. Han and Lu [1986] contains a study of the performance of a set of four evaluation strategies (including Naive and Henschen-Naqvi and two others not considered here) on the same generation example, using randomly generated data. Their model is based on the selectivity of the join and select operations and the sizes of the data relations. They consider both CPU and IO cost. We have chosen to concentrate on one aspect of the problem, which is the number of successful firings (measured using the sizes of the intermediate relations), and have studied a wider range of strategies, queries and data.

12. Conclusions and Caveats

We have presented a performance comparison of ten methods. Even though the "benchmark" we have used is incomplete, the cost measure too elementary and the approximations crude, we found the results to be valuable. The robustness of the results (at least on our workload), both in terms of the order of magnitude differences between the costs of the strategies and in terms of invariance of the results to the parameters that we varied, was a surprise. We have also been able to explain most of our results through three factors: duplication, relevant facts and unary vs. binary. While the first two factors were well known, the third one came as a surprise, even though it was probably already understood in Sacca and Zaniolo [1986].

Our conclusions may be summarized as follows:

1. For a given query, there is a clear ordering of the strategies.
2. The more specialized strategies perform significantly better.
3. Recursion is a powerful control structure which reduces the number of relevant facts and eliminates duplicate work.
4. The choice of the right strategy is critical since the differences in performance are by orders of magnitude.
5. Three factors which greatly influence performance are: (i) duplication of work, (ii) the set of relevant facts, and (iii) the number and arity of the intermediate relations.

The results seem robust in that the performance of the various strategies usually differ by orders of magnitude, which allows a wide latitude for the approximations in the model and cost evaluation. Also, the curves rarely intersect, which means that the relative ordering of the strategies is maintained in most cases over the entire range of data.

However, it must be emphasized that our cost function makes some crude approximations. The cost of join is linear in the size of the result, a consequence of our using the size of intermediate relations as the cost measure. We also ignore the cost of disk accesses, and the cost of implementing a recursive control strategy. Our model suffers from the approximation that duplication is independent of the size of the start set.

Finally, our sample data and queries are limited, and the results must be extrapolated to other data and queries with caution, especially since the results show some variance in the relative performance of the strategies for different sets of data and queries. In particular, our benchmark is limited to the type of data and query where there is a *large* amount of data and the size of the answer to the query is *small*. This clearly favors the "smart" strategies and obscures, for instance, the fact that Semi-Naive performs as well as any other strategy when computing the entire transitive closure of a relation (Bancilhon [1986]).

Further, our data contains no cycles or shortcuts. This is an important limitation since it favors some of the specialized strategies. We emphasize the importance of this limitation since it is possible to (mis-)interpret our results as evidence for the uniform superiority of the Henschen-Naqvi and Counting methods. For example, cycles in the data make Counting inapplicable. The presence of shortcuts may well make these methods considerably more expensive. For instance, there are cases where Counting performs worse than Magic Sets (Bancilhon et al. [1986a]).

13. Acknowledgements

We wish to thank Bill Alexander, Patrick Valduriez and Ken Smith for careful proofreading of parts of the manuscript. We are extremely grateful to Paris Kanellakis, Eliezer Lozinskii, Jeff Ullman, Laurent Vieille and Carlo Zaniolo who provided enlightening comments and suggested many corrections and improvements to the paper. We thank the anonymous referees of Bancilhon and Ramakrishnan [1988] for

numerous comments which improved the technical content and presentation of this paper.

14. References

1. F. Afrati, C. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv and J.D. Ullman [1986], "Convergence of Sideways Query Evaluation," *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*
2. A. Aho and J. D. Ullman [1979], "Universality of Data Retrieval Languages," *Proc. 6th ACM Symposium on Principles of Programming Languages, 1979, pp 110-120.*
3. F. Bancilhon [1986], "Naive Evaluation of Recursively Defined Relations," in *On Knowledge Base Management Systems - Integrating Database and AI Systems, Brodie and Mylopoulos, Eds., Springer-Verlag, 1986, pp 165-178.*
4. F. Bancilhon [1985], "A Note on the Performance of Rule Based Systems," *MCC Technical Report DB-022-85, 1985.*
5. F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman [1986a], "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986, pp 1-15.*
6. F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman [1986b], "Magic Sets: Algorithms and Examples," *Unpublished Manuscript, 1986.*
7. F. Bancilhon and R. Ramakrishnan [1988], "Performance Evaluation of Data Intensive Logic Programs", In *Foundations of Deductive Databases and Logic Programming, Ed. J. Minker, Morgan Kaufman, 1988.*
8. R. Bayer [1985], "Query Evaluation and Recursion in Deductive Database Systems," *Unpublished Manuscript, 1985.*
9. C. Beeri and R. Ramakrishnan [1987], "On the Power of Magic," *Proc. 6th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 1987, pp 269-283.*
10. C. Chang [1981], "On the Evaluation of Queries Containing Derived Relations in Relational Databases," In *Advances in Data Base Theory, Vol.1, H.Gallaire, J. Minker and J.M. Nicolas, Plenum Press, New York, 1981, pp 235-260.*
11. C. Delobel [1986], "Bases de Donnees et Bases de Connaissances: Une Approche Systemique a l'Aide d'une Algebre Matricielle des Relations," *Journees Francophones, Grenoble, January 1986, pp 101-134.*
12. S.W. Dietrich and D.S. Warren [1985], "Dynamic Programming Strategies for the Evaluation of Recursive Queries," *Unpublished Report, 1985.*
13. H. Gallaire, J. Minker and J.-M. Nicolas [1984], "Logic and Data Bases: A Deductive Approach," *Computing Surveys, Vol. 16, No 2, June 1984, pp 153-185.*
14. G. Gardarin and Ch. de Maindreville [1986], "Evaluation of Database Recursive Logic Programs as Recurrent Function Series," *Proc. SIGMOD 86, Washington, D.C., May 1986, pp 177-186.*
15. J. Han and H. Lu [1986], "Some Performance Results on Recursive Query Processing in Relational Database Systems," *Proc. Data Engineering Conference, Los Angeles, February 1986, pp 533-539.*
16. L. Henschen and S. Naqvi [1984], "On Compiling Queries in Recursive First-Order Data Bases," *JACM, Vol 31, January 1984, pp 47-85.*
17. M. Kifer and E. Lozinskii [1986a], "Filtering Data Flow in Deductive Databases," *Proc. International Conference on Database Theory, Lecture Notes in Computer Science, No. 243, Springer-Verlag, 1986, pp 186-202.*
18. M. Kifer and E. Lozinskii [1986b], "A Framework for an Efficient Implementation of Deductive Databases," *Proc. 6th Advanced Database Symposium, 1986, pp 109-116.*
19. M. Kifer and E. Lozinskii [1988], "SYGRAF: Implementing Logic Programs in a Database Style," *To appear, IEEE Trans. on Software Engineering, 1988.*

20. R. Krishnamurthy, R. Ramakrishnan and O. Shmueli [1988], "A Framework for Testing Safety and Effective Computability of Extended Datalog," *To appear, Proc. ACM-SIGMOD Conference, 1988.*

21. K. Laskowski [1984], "Compiling Recursive Axioms in First Order Databases," *Masters Thesis, Northwestern University, 1984*

22. E. Lozinskii [1985], "Evaluating Queries in Deductive Databases by Generating," *Proc. 11th International Joint Conference on Artificial Intelligence, 1985, pp 173-177.*

23. G. Marque-Pucheu [1983], "Algebraic Structure of Answers in a Recursive Logic Database," *To appear in Acta Informatica.*

24. G. Marque-Pucheu, J. Martin-Gallausiaux and G. Jomier [1984], "Interfacing Prolog and Relational Database Management Systems," *in New Applications of Databases, Gardarin and Gelenbe Eds, Academic Press, London, 1984.*

25. D. McKay and S. Shapiro [1981], "Using Active Connection Graphs for Reasoning with Recursive Rules," *Proc. 7th International Joint Conference on Artificial Intelligence, 1981, pp 368-374.*

26. J. Minker [1988], "Foundations of Deductive Databases and Logic Programming," *Ed. J. Minker, Morgan Kaufmann, 1988.*

27. K. Morris, J. Ullman and A. Van Gelder [1986], "Design Overview of the NAIL! System," *Proceedings of the 3rd International Conference on Logic Programming, London, July 1986.*

28. R. Ramakrishnan [1988], "Magic Templates: A Spell-Binding Approach to Logic Programs," *Technical Report, Computer Sciences Department, Univ. of Wisconsin-Madison, 1988.*

29. R. Ramakrishnan, F. Bancilhon and A. Silberschatz [1987], "Safety of Horn Clauses with Infinite Relations," *Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987.*

30. J. Rohmer, R. Lescoeur and J.M. Kerisit [1986], "The Alexander Method: A Technique for the Processing of Recursive Axioms in Deductive Databases," *New Generation Computing 4, 3, 1986, pp 273-285.*

31. A. Rosenthal, S. Heiler, U. Dayal and F. Manola [1986], "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," *Proc. ACM-SIGMOD Conference, 1986.*

32. P. Roussel [1975], "PROLOG, Manuel de Reference et de Utilisation," *Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1975.*

33. D. Sacca and C. Zaniolo [1986a], "On the Implementation of a Simple Class of Logic Queries for Databases," *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986, pp 16-23.*

34. D. Sacca and C. Zaniolo [1986b], "The Generalized Counting Method for Recursive Logic Queries," *Proc. First International Conference on Database Theory, 1986.*

35. D. Sacca and C. Zaniolo [1987], "Magic Counting Methods," *Proc. ACM-SIGMOD Conference, 1987, pp 49-59.*

36. S. Shapiro and D. McKay [1980], "Inference with Recursive Rules," *Proc. 1st Annual National Conference on Artificial Intelligence, August, 1980.*

37. S. Shapiro, J. Martins and D. McKay [1982], "Bi-Directional Inference," *Proc. 4th Annual Conference of the Cognitive Science Society, Ann Arbor, Michigan, 1982.*

38. A. Tarski [1955], "A Lattice Theoretical Fixpoint Theorem and its Applications" *Pacific Journal of Mathematics 5, 1955, pp 285-309*

39. J.D. Ullman [1985], "Implementation of Logical Query Languages for Databases," *Transactions on Database Systems, Vol. 10, No. 3, 1985, pp 289-321.*

40. J. Ullman and A. Van Gelder [1985], "Testing Applicability of Top-Down Capture Rules," *Technical Report, Stanford University, STAN-CS-85-1046, 1985.*

41. P. Valduriez and H. Boral [1986], ''Evaluation of Recursive Queries Using Join Indices,'' *Proc. First Intl. Conference on Expert Database Systems, Charleston, 1986.*

42. M. Van Emden and R. Kowalski [1976], ''The Semantics of Predicate Logic as a Programming Language,'' *JACM, Vol 23, No 4, October 1976, pp 733-742.*

43. A. Van Gelder and R. Topor [1987], ''Safety and Correct Translation of Relational Calculus Formulas,'' *Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987.*

44. L. Vieille [1986], ''Recursive axioms in Deductive Databases: The Query/Subquery Approach,'' *Proc. First Intl. Conference on Expert Database Systems, Charleston, 1986, pp 179-194.*

45. L. Vieille [1988], ''From QSQ towards QoSaQ: Global Optimization of Recursive Queries,'' *To appear in "Proc. 2nd Intl. Conf. on Expert Database Systems", Ed. L. Kerschberg, 1988.*

46. C. Zaniolo [1985], ''The Representation and Deductive Retrieval of Complex Objects,'' *Proc. 11th Int. Conference on Very Large Data Bases, Stockholm, September 1985.*

47. C. Zaniolo [1986], ''Safety and Compilation of Non-Recursive Horn Clauses,'' *Proc. First Intl. Conference on Expert Database Systems, Charleston, 1986.*