12 Rule Transformation Methods in the Implementation of Logic Based Languages

DOMENICO SACCA[1]
University of Calabria
Rende, Italy

CARLO ZANIOLO
MCC
Austin, Texas

1. Introduction

Several novel techniques have been developed to support the efficient implementation of Logic based languages oriented towards data intensive applications, such as LDL (Tsur and Zaniolo, 1986; Beeri et al., 1987) and NAIL! (Morris et al., 1987) and several others (Zaniolo, 1987). A crucial problem in the implementation of these languages is the efficient support for recursion. Among the most significant techniques developed for the solution of this problem, we find the Differential (Seminaive) Fixpoint Method (Balbin and Ramamohanarao, 1987; Saccà and Zaniolo, 1988), the Magic Set Method (Bancilhon et al., 1986), the Minimagic Method (Saccà and Zaniolo, 1987), the Counting Method (Saccà and Zaniolo, To appear) and the Magic Counting Method (Saccà and Zaniolo, 1986; 1987).

[1]Part of this work was done while the author was visiting at MCC. This author was also supported by a grant of C.N.R. (project "Sistemi evoluti per Basi di Dati").

These techniques were used in the implementation of the LDL system that was designed to support and manage efficiently large data and rule sets (Chimenti et al., 1987), and construct all-answer solutions to queries more efficiently than current Prolog implementations. Thus, LDL is a powerful database languages, which removes the "impedance mismatch" between the query language and the host programming language, currently besetting the development of database applications.

Take for instance the recursive predicate SG of Fig. 1,

$$r_o{:}SG(x,y): -P(x,x1), SG(x_1,y_1), P(y,y_1)$$
$$r_1{:}SG(x,x): -H(x).$$

Fig. 1. The same-generation example.

where $P(x,x_1)$ is a previous parent predicate (either defined as a *parent* database relation or as the union of two database relations *mother* and *father*) and $H(x)$ is a database predicate denoting all humans. Non recursive rules over a fact base can be supported efficiently in a bottom up fashion by an extended relational algebra (Zaniolo, 1985) implementing the *immediate consequence* operator T_P (Lloyd, 1984). In conformity with the bottom up, matching-based execution strategy and the operator-based approach we want to implement recursive queries, by a least Fixpoint operator (Van Emden and Konolski, 1976).

For instance, for a program P consisting of the SG rules and the query,

$$G1{:}\ ?SG(x,y) \tag{1}$$

one will start by setting the initial value of a variable relation SG to empty and computing a new value, say

$$\mathbf{SG}' = \mathbf{SG} \cup T_P(\mathbf{SG}) \tag{2}$$

where T_P denotes, e.g., the relational algebra equivalent to the SG rules (Ullman, 1982):

$$T_P(\mathbf{SG}) = \pi_{1,1}\mathbf{H} \cup \pi_{1,5}((\mathbf{P} \bowtie_{2=1} \mathbf{SG}) \bowtie_{4=2} \mathbf{P})). \tag{3}$$

Thus, after replacing $\mathbf{SG}$ by $\mathbf{SG}'$, the computation iterates and terminates when $\mathbf{SG}' = \mathbf{SG}$. Since our rules contain only positive goals, the function T_P is monotonic and continuous in the lattice of relations defined by set containment, and this procedure computes a unique least solution of the Fixpoint equation $\mathbf{SG} = T_P(\mathbf{SG})$ (Tarski, 1955).

This naive Fixpoint approach suffers from two problems. One is the fact that some computation is repeated unnecessarily— for instance the naive Fixpoint algorithm will recompute the first level ancestors of a pair (a, b) at the first step of the Fixpoint iteration and then again at each step that follows. Differential techniques (Paige and Koenig, 1982) used to solve this problem, yielding the differential (alias seminaive) Fixpoint method discussed in the next section. The second problem occurs when a recursive predicate is called with some arguments bound. For instance, for the query,

$$G2\text{: } ?SG(adam, y) \tag{4}$$

the Fixpoint approach will still compute all possible pairs of humans that are of the same generation, only to later discard those that do not have "adam" as their first argument. Clearly, we need a strategy for taking advantage of the constants present in the query. Unfortunately, the simple approach of specializing the rules by substituting the constants in place of the variables does not work for recursive predicates. For instance, if we replace the occurrence of the variable x by "adam" we obtain rules that do not produce humans of the same generation as adam. (Of course, there are also cases in which the substitution trick works (Aho and Ullman, 1979); but recognizing those cases is, in general, undecidable (Beeri et al., 1987).) Thus, we need novel techniques for taking advantage of constants in recursive queries. Such a need is underscored by the safety issues discussed next.

To illustrate a first aspect of the safety issue let us assume that the only goal (i.e., $H(x)$) is removed from rule r_1 in Fig. 1. Then, assuming an infinite underlying universe, the query $G1$ becomes unsafe, since any pair (a, a), with a an arbitrary constant would satisfy it. The query $G2$ is safe, per se, since only a finite number of people can be of the same generation as adam; but its evaluation using the Fixpoint approach is still unsafe and, therefore, unfeasible. To solve this problem we need an execution plan capable of taking advantage of instantiated arguments in the recursive goal.

The safety issue is even more of a concern for "computational" predicates such as that of Fig. 2, which are normally intended for use only with certain input bindings. The potential sources of unsafe behavior in the rules of Fig. 2 are two. One is that the non-recursive rules r_2 and r_3 define MG to be satisfied by infinitely many pairs of x values—this is the same problem as in the previous example. However, even if the x-values range over a finite set, the recursive rules r_0 and r_1 generate longer and longer lists at each step in the Fixpoint computation, which, therefore, never ends (unsafe computation).

r_0: $MG(x \bullet y, x_1 \bullet y_1, x \bullet w) : -MG(y, x_1 \bullet y_1, w), x \geq x_1$
r_1: $MG(x \bullet y, x_1 \bullet y_1, x_1 \bullet w) : -MG(x \bullet y, y_1, w), x < x_1$
r_2: $MG(\mathbf{nil}, x, x)$
r_3: $MG(x, \mathbf{nil}, x)$

Fig. 2. Merging two sorted lists.

In conclusion, an effective usage of the instantiated arguments in a recursive goal is vital for performance reasons and to avoid the non-termination pitfall. Therefore, it should be of no surprise that a number of interesting approaches were proposed to deal with this problem, including (Bancilhon et al., 1986; Beeri and Ramakrishnan, 1987; Gardarin and De Maindreville, 1986; Chandra and Harel, 1982; Henschen and Naqvi, 1984; Kifer and Lozinskii, 1986; McKay and Shapiro, 1981; Saccà and Zaniolo, 1986; Ullman, 1985; Vieille, 1986). The reader is referred to Bancilhon and Ramakrishnan (1986) for an overview and a comparison of these approaches. As described in Bancilhon and Ramakrishnan (1986), some of these approaches lack generality—i.e., the realm in which they work is either limited or not clearly understood—and they lack robustness—i.e., they are formulated as a sequential algorithm for main memory based execution —hence their compatibility with different execution models and architectures is unclear.

In this paper, we propose a comprehensive rule rewriting approach and supporting algorithms on which the actual implementation of LDL is based (Chimenti et al., 1988). The outline of the paper is as follows:

In Section 2 we present the differential (alias seminaive) Fixpoint algorithm.

In Sections 3 and 4 we turn to the implementation of recursive predicates when some arguments in the recursive goal are instantiated. We describe two methods, *the magic set method and the counting method.* These rewrite the original rules, that, as such, cannot be implemented safely and efficiently as a single Fixpoint, into equivalent ones that can be implemented by a pair of (seminaive) Fixpoint computations safely and efficiently.

In Section 5, we consider the *specialized single Fixpoint method* that handles various cases of practical import where one of the two Fixpoint computations can be eliminated.

In Section 6, we compare the pros and cons of various methods and study their computational complexity.

In Section 7 we introduce the magic counting method that combines the strong features of the previous methods.

2. Seminaive Fixpoint Algorithm

In a logic program LP, a predicate P is said to *imply* a predicate Q, written $P \rightarrow Q$, if there is a rule in LP having predicate Q as its head and predicate P in its body, or there exists a P' where $P \rightarrow P'$ and $P' \rightarrow Q$ (transitivity). Then, a predicate P will be called *recursive* when $P \rightarrow P$. Two predicates P, and Q are called *mutually recursive* if $P \rightarrow Q$ and $Q \rightarrow P$. Then, the sets of all predicates in LP can be divided into recursive predicates and non-recursive ones (such as database predicates). This implication relationship can also be used to partition the recursive predicates into disjoint subclasses of mutually recursive predicates, which we will call *recursive cliques* (having the graph representation of $\rightarrow$, in mind). All predicates in the same recursive clique will have to be computed together, they cannot be computed separately. Say that $\mathbf{R}$ denotes the vector of predicates in our recursive clique. Then, a rule defining a predicate in $\mathbf{R}$ will be called *recursive* if it contains a predicate in $\mathbf{R}$ among its goals, and it will be called an *exit rule* otherwise (e.g., in Fig. 2, r_0 and r_1 are recursive rules, while r_2 and r_3 are exit rules). A recursive rule that only contains one occurrence of a recursive predicate in $\mathbf{R}$ in its body is called *linear*. The recursive rules in Figs. 1 and 2 are linear.

Say now that $\mathbf{E}$ defines the value of the relations in $\mathbf{R}$ according to the exit rules, and $T_P(\mathbf{R})$ is the mapping defined by the recursive rules (e.g., the relational algebra equivalent of the recursive rules). Then, the Fixpoint algorithm can be formulated as follows:

```
Naive Fixpoint Algorithm
  begin
  $j =: 0$;
  $\mathbf{R} := \phi$;
  $\mathbf{R}' := \mathbf{E}$;
  while $\mathbf{R}' \neq \mathbf{R}$ do
        begin
        $\mathbf{R} := \mathbf{R}'$;
        $\mathbf{R}' := T_P(\mathbf{R}) \cup \mathbf{R}$;
        $j := j + 1$
        end
  end.
```

Fig. 3. A naive formulation of the Fixpoint algorithm.

The value **R** returned by this new algorithm is the least Fixpoint solution of the recursive equation $\mathbf{R} := T_P(\mathbf{R}) \cup \mathbf{E}$, which, in turn, defines the minimum model for the given recursive rules and exit rules defining **R**. In the algorithm of Fig. 3, we also have include a step counter j, which, although not needed at this point, will become useful later.

The problem with the naive Fixpoint algorithm is that tuples are produced over and over again. For instance, in the same generation example of Fig. 1, the parents of a given human generated in the first iteration step are generated again in the second step, where grandparents are first computed. Differential techniques have been proposed by several authors to solve the problem, including Bancilhon (1985) and Balbin and Ramanohanarao (1987). Our approach, similar to that proposed in Balbin and Ramanohanarao (1987), has the advantage of being general and not requiring differentiation of relational algebra expressions.

Assume that $\delta\mathbf{R}$ denotes the "new" tuples obtained at each step of the algorithm of Fig. 3. Then the previous algorithm can be modified into the one of Fig. 4. The final answer of the seminaive Fixpoint algorithm, i.e., **R** after a step where that $\delta\mathbf{R} = \phi$, is obviously identical to the answer **R** produced by the naive Fixpoint algorithm. This second algorithm is, however, more efficient than the previous one since we will next recast the computation $\delta\mathbf{R} := T_P(\mathbf{R}) - \mathbf{R}$ into a form that avoids the recomputation of old results.

```
Seminaive Fixpoint Algorithm
    begin
    $j := 0$;
    $\delta\mathbf{R} := \phi$;
    $\mathbf{R} := \mathbf{E}$;
    while $\delta\mathbf{R}_j \neq \phi$ do
        begin
        $\delta\mathbf{R} := T_P(\mathbf{R}) - \mathbf{R}$;
        $\mathbf{R} := \mathbf{R} + \delta\mathbf{R}$;
        $j := j + 1$
        end
    end.
```

Fig. 4. The seminaive formulation for the Fixpoint algorithm.

While previous authors have proposed rather cumbersome transformations on the relational algebra expression for $F(\mathbf{R})$, we will propose here a more direct and expressive rule rewriting approach.

For the Same Generation example the above seminaive Fixpoint algorithm could be expressed as follows:

```
begin
$j := 0$;
$SG(0, x, x) : -H(x)$;
while $SG(j, \_, \_)$ do
$SG(j + 1, x, y) : -P(x, x_1), SG(j, x_1, y_1), P(y, y_1),$ not $(SG(\_, x, y))$;
    $j := j + 1$
    end
end.
```

Fig. 5. Seminaive Fixpoint for the same generation example with an Unimproved Iterative Step.

Some remarks about the notation are needed here. First of all, observe that $J+1$ is just a short-hand notation of convenience, inasmuch as in a proper Prolog program $J+1$ would be replaced by $J1$ and the goal "$J1$ *is* $J+1$" would be added to the rule. Also observe that we are now treating our rules as those of production systems, inasmuch as we use them to updated our current computation state. Indeed, we have a three column relation SG to which we initially assign a value derived according to the exit rule, and, at each step of the Fixpoint iteration, we augment the current SG according to the recursive rule. Note that $SG(\_, x_1, y_1)$ will return the x_1, y_1 projection of all the tuples produced so far in the Fixpoint iteration—i.e., the **R** relation is updated implicitly, thus eliminating the need for the explicit step $\mathbf{R} := \mathbf{R} + \delta\mathbf{R}$ in the algorithm of Fig. 4. Thus, $not(SG(\_, x, y))$ amounts to a set difference—i.e., to the $-\mathbf{R}$ operation above. Finally, $\delta\mathbf{R} \neq \phi$ holds whenever $SG(j, \_, \_)$ is true. Thus, Fig. 5 is in fact a faithful restatement of the seminaive Fixpoint algorithm in the production rule notation.

Furthermore, one need not write the algorithm of Fig. 5. explicitly, since the application of the Naive Fixpoint of Fig. 3 to the pair of rules that follows tantamounts to the algorithm of Fig. 5.

$$\begin{aligned}&SG(0, x, x) : -H(x).\\&SG(J+1, x, y) : -P(x, x_1), SG(J, x_1, y_1), P(y, y_1),\\&\qquad \text{not } (SG(\_, x, y)).\end{aligned}\tag{5}$$

By now, our reader has probably begun to grasp how the rule rewriting approach works. We have a program P that we would like to implement with a seminaive algorithm when only the basic naive algorithm is at hand. Then, we transform P into an *equivalent* P' such that the naive algorithm on P' behaves as the seminaive on P.

The program above is only a first step, inasmuch as it suffers from two of problems. The first is that, because of the presence of negation it does not have a unique minimal model; furthermore, the result of the Fixpoint algorithm may not yield least solution of the Fixpoint equation $R = T_P(R)$—which in fact may not have a least solution (Kolaitis and Papadimitriou, 1988). Without a clear Fixpoint-based or model-theoretic semantics a claim of equivalence preserving methods would have no basis. To solve this problem we observe that we can rewrite these rules as

$$\begin{aligned}&G(0, x, x) : -H(x).\\&SG(J+1, x, y) : -P(x, x_1), SG(J, x_1, y_1), P(y, y_1),\\&\qquad \text{not } (SG(I, x, y), I \leq J).\end{aligned}\tag{6}$$

The program so transformed is *locally stratified* (Przymusinski, 1987). A program P is said to be locally stratified when its Herbrand Base H can be partitioned into strata $H_0, H_1, \ldots$ such that for every instantiation r of each rule in P, the negated goals of r

belong to strata strictly lower than that of the head of r, while the positive goals of r belong to strata not higher than that of the head of r. Programs that are locally stratified have a well-defined model theoretic semantics, based on the concept of *perfect model* (Przymusinski, 1987). Furthermore, the locally stratified programs produced by our seminaive methods, also have a least-Fixpoint based semantics (Saccà and Zaniolo, 1988). This provides a formal basis to the claim that our transformed program is equivalent to the original one.

Let us now turn to the second problem, which is that of performance. To derive the seminaive improvement we can observe, that, at step j, every value of $SG(J, x_1, y_1)$ with $J < j$ had already been used at previous steps of the Fixpoint computation and can be dropped. Thus, the previous rules can now be simplified into those of Fig. 6.

$$\begin{aligned}
&SG(0,x,x)\ :-\ H(x).\\
&SG(j+1,x,y) : -P(x,x_1), SG(j,x_1,y_1), P(y,y_1),\\
&\qquad \text{not } (SG(I,x,y), I \le j).
\end{aligned}$$

Fig. 6. Seminaive Fixpoint for the same generation example with an Improved Iterative Step.

where the j value to be used (in the relational algebra based implementation of these rules) is that of the running index of the Naive Fixpoint of Fig. 3. The improvement of the iterative step in the seminaive computation is thus straightforward for linear rules.

Non-linear rules require a more complex rewriting, which is discussed next. In general, each non-linear rule in a recursive clique has the form

$$r\colon P : -D_0, P_1, D_1, P_2, \ldots, P_n, D_n, \tag{7}$$

where $P_1, \ldots, P_n$ denote the *occurrences of the recursive predicates* from the clique, $(0 \le i \le n)$, and $D_0, \ldots, D_n$ denote non-recursive predicates or (possibly empty) conjunctions thereof. Let $P[j+1]$, $P[j]$ and $P[I_i]$, respectively, denote the predicate P expanded by the addition of a first argument $j+1$, j and "I_i," then our rule r is expanded into a disjunction of n rules to be used in the seminaive algorithm template as follows:

$$\begin{aligned}
r_1\colon P[j+1] :- & D_0,\ P_1[j], D_1, P_2[I_2], \ldots, P_n[I_n], D_n \\
& I_2 \le j, \ldots, I_n \le j, \text{not}\,(P[I], I \le j). \\
r_2\colon P[j+1] :- & D_0, P_1[I_1], D_1, P_2[j], \ldots, P_n[j], D_n \\
& I_1 \le j, \ldots, I_n \le j, \text{not}\,(P[I], I \le j). \\
& \vdots \\
r_2\colon P[j+1] :- & D_0, P_1[I_1], D_1, P_2[I_2], \ldots, P_n[j], D_n \\
& I_2 \le j, \ldots, I_{n-1} \le j, \text{not}\,(P[I], I \le j).
\end{aligned} \tag{8}$$

For instance, for the non-linear formulation of ancestors, of Fig. 7,

$$\begin{aligned}
&Anc(x,y) : -P(x,y). \\
&Anc(x,z) : -Anc(x,y), Anc(y,z).
\end{aligned}$$

Fig. 7. Non-linear formulation for ancestors.

The rules resulting from the seminaive method are given in Fig. 8.

$$\begin{aligned}
&Anc(0,x,y) : -P(x,y); \\
&Anc(j+1,x,z) : -Anc(j,x,y), Anc(I_2,y,z), I_2 \le j, \\
&\quad not(Anc(I,x,z), I \le j). \\
&Anc(j+1,x,z) : -Anc(I_1,x,y), Anc(j,y,z), I_1 \le j, \\
&\quad not(Anc(I,x,z), I \le j).
\end{aligned}$$

Fig. 8. Seminaive improvement for the example of Fig. 7.

3. The Magic Set Method and the Minimagic Method

As previously discussed, it is often the case that constants cannot be migrated into recursive rules. For these cases however, a small set of values, called a *magic set*, can be derived from these constants and used to restrict the Fixpoint computation and make it either safe or more efficient. This observation is at the basis of the *magic set method* introduced in Bancihon er al. (1986). For example, in the query of Fig. 9,

$$
\begin{aligned}
&G5{:}?SG(adam, y)\\
&r_0{:}SG(x, y) : -P(x, x_1), SG(x_1, y_1), P(y, y_1)\\
&r_1{:}SG(x, x) : -H(x).
\end{aligned}
$$

Fig. 9. Find persons of the same generation as adam.

we can restrict our search to the values of x in r_1 and to the values of x and x_1 in r_0 denoting persons that are ancestors of adam. Thus, the set of ancestors of adam is the magic set for the query at hand. From an operational viewpoint, magic set values are those instantiated by Prolog during the goal expansion phase (first phase) of the SLD resolution. During this phase, r_0 is invoked first and the x value is bound to "adam" while x_1 is bound to a parent of adam. As rule r_0 is called again, with x now bound to a parent of adam, x_1 becomes instantiated to a grandparent of adam, and so on. Also observe that y_1 and y are not instantiated during this phase; these variables become instantiated during the second (tree reduction) phase, which begins after new ancestors of adam run out and rule r_1 is used. Therefore, the application of the magic set method involves three steps.

Step 1: The *binding passing analysis* determines which arguments in the recursive predicates will be instantiated during the first phase of the SLD resolution,

Step 2: The *rule rewriting step* adds rules for computing the magic sets and modifies the original rules to take advantage of these magic sets, and

Step 3: The *Fixpoint-based execution* of the rules produced in Step 2 constructs the answer.

Let us illustrate these steps on the example of Fig. 9. The binding passing analysis will determine that

(a) the first argument in the recursive predicate is bound by the query, and
(b) given that the first argument (i.e., x) is instantiated in the head of the recursive rule, then, the first argument in the recursive predicate of the body (i.e., x_1) can also be instantiated via the goal $P(x, x_1)$.

Thus, we have a cyclic situation illustrated by the graph of Fig. 10 (also known as a *binding graph* (Saccà and Zaniolo, 1987)), where SG with a bound first argument (denoted by SG^1) in turn binds the first argument of SG again.

$$SG^1$$

Fig. 10. Binding passing graph for the example of Fig. 9.

Therefore, for the example in Fig. 9, the first argument of SG is always instantiated during the first phase of the SLD resolution, and thus the magic set will collect the various values taken by this argument during this phase.

The binding passing analysis (Step 1) can be done at query compilation time since it only depends on the intentional information (query and rules). Step 2 can also be performed at compilation time since it operates only on the intentional database by adding rules defining a magic predicate and modifying the original recursive rules with the addition of the magic predicate as a goal to act as a filter. However, the magic set computation depends on the extensional information and must be done at execution time.

For the Same Generation example, for instance, the magic rules are given in Fig. 11.

%magic rules—ancestors of adam
$m.SG(adam)$
$m.SG(x_1) : -P(x, x_1), m.SG(x).$

%modified rules
$SG'(x, y) : -m.SG(x), H(x).$
$SG'(x, y) : -m.SG(x), P(x, x_1), SG'(x_1, y_1), P(y, y_1).$

%modified query
$?SG'(adam, y)$

Fig. 11. Magic rules and modified rules for the query of Fig. 9.

Note that $m.SG(x)$ contains the set of all ancestors of adam, as expected. Therefore, the addition of the goal $m.SG(x)$ to the original rules yields a much more constrained (i.e., more efficient) computation for SG'. The starting point of this computation (as per the exit rule) no longer consists of the set of all humans, but rather the set of ancestors of adam—thus, the goal $H(x)$ could be removed without making the query unsafe. Moreover, owing to the $m.SG$ goal in the modified recursive rule, persons who are not ancestors of adam are discarded at each step of the Fixpoint computation of SG'.

Therefore, the result of the magic set method is a transformed program featuring (i) a new recursive clique containing the magic predicates and (ii) the old recursive clique containing the old recursive predicates with modified rules. Every recursive clique is translated into a Fixpoint computation; therefore, the original program, that is unsuitable to implementation via a single Fixpoint computation, has been transformed into an equivalent one that is amenable to implementation by a pair of Fixpoint computations. These two Fixpoint computations are then further optimized using the seminaive improvement described in the last section.

The reader can refer to Saccà and Zaniolo (1987) for a more complete and formal description of the magic set algorithm, including the binding passing analysis which determines the arguments in the recursive rules that are instantiated by the query constants. Of more direct interest in our discussion are the rewriting rules used to implement the magic set method.

3.1. Rewriting Rules for the Magic Set Method

3.1.1. Exit Magic Rule

There is exactly one such a rule and it consists of a unit clause constructed from the query goal by eliminating non-constant arguments and adding the distinguished prefix "*m.*" to the predicate name.

3.1.2. Recursive Magic Rules

They are derived from the recursive rules as follows:

Case 1. The given rule is linear. Then do the following:

(a) drop the goals with no instantiated arguments(e.g., $P(y, y_1)$ is dropped for the example of Fig. 9),
(b) drop the non-instantiated arguments from the recursive goals, (y and y_1 are dropped for the example of Fig. 9)

(c) exchange the recursive predicate in the head with the recursive predicate in the tail; these predicates are also renamed with the prefix "$m.$" (see Fig. 11 for the case at hand).

Case 2. The given rule is non linear, with $n > 1$ occurrences of recursive predicates in the body. Then we do the following:

(a) Construct n linear rules; each such a rule is obtained by deleting all occurrences of the recursive predicates but one (obviously, a different one for each rule).
(b) Construct a magic rule from each of the n rules so obtained, using the transformation indicated in Case 1.

3.1.3. Building the Modified Rules

Each original rule is modified by the addition of the corresponding magic predicate. This has the form $m.P(X)$, with P the predicate symbol in the head of the rule and X the bound arguments in said head (For the example of Fig. 10, we add $m.SG(x)$).

Observe that there is nothing magic about the particular value adam appearing in the query of Fig. 9, since the solution just described applies to any goal $SG(z, y)$ provided that z was instantiated, say, by previous goals in the rule. All that is needed is the ability to pass this instantiated value in the exit magic rule of Fig. 11. Thus, the magic set method—and, in general all the methods described in this paper—can also be used to support sideways information passing into recursive predicates.

The magic set method was first proposed in Bancilhon et al. (1986) for linear rules; it was then generalized in Saccà and Zaniolo (1986) to deal with function symbols and certain non-linear rules and further extended in Beeri and Ramakrishnan (1987) with a notion of sideway information passing making it applicable to arbitrary non-linear rules.

The magic set method is elegant and general, but suffers from two drawbacks. Concerning the first drawback, observe that the natural way to support the modified rule of Fig. 11 consists of joining the current relation $SG'(x_1, y_1)$ with $P(x, x_1)$ and then with $m.SG(x)$. However, if we assume that $P(x, x_1)$ is conditionally safe, i.e., can be evaluated only when the first argument is instantiated (typically, arithmetic predicates and recursive predicates are conditionally safe) then, the only way to support the modified rules of Fig. 11, is to join (the relation corresponding to) $m.SG(x)$ with $P(x, x_1)$ first, and then join the result with $SG'(x_1, y_1)$ and, finally, with $P(y, y_1)$. The second drawback is that the join of $m.SG(x)$ with $P(x, x_1)$ is exactly the computation performed in building the magic set. Thus, rather than repeating this computation at each step of the second Fixpoint iteration, we can save the result of the first Fixpoint computation and use it in performing the second one. This observation is at the basis of the *minimagic method,* (Saccà and Zaniolo, 1986) where all the results of the first Fixpoint computation are saved in a "magic relation". This relation is then used in the modified rule computation in lieu of the original predicates and solved goals. Thus, in the example in question, the two goals $m.SG(x)$ and $P(x, x_1)$ would be replaced by a goal $m.P(x, x_1)$ which is basically the restriction of the parent relation over the ancestors of "adam". In general, the minimagic method avoids duplicate computation at the price of storing larger data sets.

The magic set transformation for the list-merge example of Fig. 2 and the query

$$?MG(L_1, L_2, w) \tag{9}$$

is given in Fig. 12.

$m.MG(L1, L2).$

$m.MG(y, x_1 \bullet y_1) : -m.MG(x \bullet y, x_1 \bullet y_1), x \geq x_1$
$m.MG(x \bullet y, y_1) : -m.MG(x \bullet y, x_1 \bullet y_1), x < x_1$

$MG'(x \bullet y, x_1 \bullet y_1, x \bullet w) : -m.MG(x \bullet y, x_1 \bullet y_1),$
$\quad MG'(y, x_1 \bullet y_1, w), x \geq x_1$
$MG'(x \bullet y, x_1 \bullet y_1, x_1 \bullet w) : -m.MG(x \bullet y, x_1 \bullet y_1),$
$\quad MG'(x \bullet y, y_1, w), x < x_1$
$MG'(\mathbf{nil}, x, x) : -m.MG(\mathbf{nil}, x).$
$MG'(x, \mathbf{nil}, x) : -m.MG(x, \mathbf{nil}).$
$?MG'(L_1, L_2, w).$

Fig. 12. The magic set method for the list merge example.

Fig. 12 illustrates the problem inflicted upon the magic set method by predicates that are only conditionally safe. Here the two comparison predicates, $x \geq x_1$ and $x < x_1$, which were used for computing the magic set, can not be used in the modified rules to derive new values of x or x_1. Thus we have to start the computation of the modified rule from the $m.MG$ relation rather than from the smaller MG'.

For this particular example, the minimagic method would use no additional storage, with respect to the magic set method, since it does not require saving the values of any additional variables. Moreover, the minimagic method would avoid duplication of computation by dropping the goal, $x \geq x_1$ ($x < x_1$) from the first (second) modified rule and using two distinct magic relations—one for the first rule and one for the second.

In summary, we have presented the magic set method and also discussed the minimagic method that improves on the former by avoiding the repetition of computation. However, we will not discuss the minimagic method any further since we want to present next the counting method, that eliminates duplication of computation even further. As an historical note, however, we would like to mention the similarity of the minimagic method (Saccà and Zaniolo, to appear) with the Alexander method described in Rohmer et al. (1986). While the Alexander method predates both the magic method and the minimagic method, it was, at first, overlooked in the research community and in topical surveys (an unfortunate event attributable, at least in part, to inadequate documentation). More recently, the method has become better known through the works of Beeri and Ramakrishnan (1987) and Demo et al. (1986).

4. The Counting Method

The Counting Method eliminates the duplicate computation of the magic set method by using counting indices. For instance, the humans of the same generation as adam can be found by first computing the ancestors of adam and then the descendants of these ancestors—provided that the levels of these ancestors and their descendants are also computed and compared as follows. Starting from adam, who is a zero-level ancestor of himself, we increase the count by one every time the recursive rule is used to generate a set of new ancestors. Symmetrically, the descendants of these ancestors are computed, by decreasing the count at each step in the recursive computation; once we obtain a descendant of level zero we can add this name to the query answer. Figure 13 gives the counting method code for the query of Fig. 9 ($j+1$ and $j-1$ are again shorthand notations of convenience similar to that used in Section 2).

%counting rules—counting up on ancestors
$cnt.SG(0, adam)$
$cnt.SG(j+1, x_1) : -P(x, x_1), cnt.SG(j, x).$

%modified rules—counting down on descendants
$SG'(j, x) : -cnt.SG(j, x), H(x).$
$SG'(j-1, y) : -SG'(j, y_1), P(y, y_1).$

%modified query
$?SG'(0, y)$

Fig. 13. Counting rules and modified rules for example of Fig. 9.

The predicate *cnt.SG* computes the ancestors of adam and counts the levels up. The predicate SG' computes the descendants of these ancestors and counts the levels down. The modified query selects the descendants whose associated index is zero and, thus, are of the same generation as adam.

Figure 13 also illustrates the rewriting rules used to implement the counting method. The counting sets are generated in a fashion similar to that of the magic rules; the main differences lie in the introduction of a counting index to record the level. Since a new level is introduced at each step, the seminaive Fixpoint computation of the counting sets can be simplified by removing the duplicate elimination check. Moreover, the modified rules are much simpler than those of the magic set method, since they do not contain the goals previously solved in computing the counting sets. Thus, the counting method avoids duplication of computation.

The idea of counting was first introduced in Bancilhon et al. (1986) and formalized in Saccà and Zaniolo (1986) for a simple class of linear queries. The method was then generalized to more complex situations in Saccà and Zaniolo (1987), using the rule-rewriting approach discussed next.

The *Generalized Counting Method* (Saccà and Zaniolo, 1987) makes use of additional indices and *supplementary counting sets.* The additional indices are required when there is more than one recursive rule defining a recursive predicate or when there are non-linear rules. The supplementary counting set is used to save values that will then be needed during the second phase. Figure 14, for instance, gives the generalized counting method for the query $?MG(L_1, L_2, w)$.

%Counting Rules:
$?cnt.MG(0, 0, L1, L2).$

$cnt.MG(j+1, 2*k+0, y, x_1 \bullet y_1) : -cnt.MG(j, k, x \bullet y, x_1 \bullet y_1),$
    $x \geq x_1$
$cnt.MG(j+1, 2*k+1, x \bullet y) : -cnt.MG(j, k, x \bullet y, x_1 \bullet y_1),$
    $x < x_1$

%Supplementary Counting Rules:
$spcnt.MG.1(j, k, x) : -cnt.MG(j, k, x \bullet y, x_1 \bullet y_1), x \geq x_1$
$spcnt.MG.2(j, k, x_1) : -cnt.MG(j, k, x \bullet y, x_1 \bullet y_1), x < x_1$

%Modified Rules:
$MG'(j-1, (k-0)/2, x \bullet w) : -spcnt.MG.1(j-1, k/2, x),$
    $MG'(j, k, w)$
$MG'(j-1, (k-1)/2, x_1 \bullet w) : -spcnt.MG.2(j-1, (k-1)/2, x_1),$
    $MG'(j, k, w)$
$MG'(j, k, x) : -cnt.MG(j, k, \mathbf{nil}, x).$
$MG'(j, k, x) : -cnt.MG(j, k, x, \mathbf{nil}).$

%Modified Query:
$?MG'(0, 0, w).$

Fig. 14. The counting method on the list merge example.

The integer arithmetic performed on the second index k is such that the complete history of the rules used in the counting phase are preserved and can be recovered in the second phase.

The counting method has two strong points and a weak one. The first plus for the method is represented by its performance that has been shown to be superior to that of other methods, both in terms of typical behavior (Bancilhon and Ramakrishnan, 1986) and worst case behavior (Marchetti-Spaccamela et al., 1987). The second plus is that it provides a framework to identify many cases of practical interest where one the two Fixpoint computations becomes unnecessary and the method can be further improved. On the minus side there is the potentially unsafe behavior of the method in the presence of cycles in the database, and problems with managing the supplementary indices needed for arbitrary depths of recursion. We will discuss these three issues next, beginning with the cases in which the counting method reduces to a single Fixpoint computation.

5. Specialized Single Fixpoint

In many practical situations, the computation plan prescribed by the magic set method or the counting method is unnecessarily complex. These are the cases in which the given recursive query with constants can be supported efficiently via a single Fixpoint computation. A typical situation is the computation of a transitive closure, such as for the linear ancestor example of Fig. 15:

$$\begin{aligned}
&anc(x, x).\\
&anc(x, z) : -anc(x, y), P(y, z).
\end{aligned}$$

Fig. 15. A right linear formulation of ancestor.

$$\begin{aligned}
&?anc(john, x).\\
&cnt.anc(0, john).\\
&cnt.anc(j + 1, x) : -cnt.anc(j, x).\\
&anc'(j, x) : -cnt.anc(j, x).\\
&anc'(j - 1, z) : -anc'(j, y), P(y, z).\\
&?anc'(0, x).
\end{aligned}$$

Fig. 16. Counting method for query (5.1) on rules of Fig. 15.

Since the indices are no longer needed in the modified rules as shown in Fig. 16, the counting rules become useless and can be eliminated. In short, the computation reduces to the program of Fig. 17:

$$\begin{aligned}
&anc'(john).\\
&anc'(z) : -anc'(x), P(y, z).\\
&-?anc'(x).
\end{aligned}$$

Fig. 17. Specialized single Fixpoint program for query (5.1).

Thus the computation reduces to a single Fixpoint on a unary relation.

Consider now the second query (5.2) on the same set of rules of Fig. 15.

$$?anc(x, mark) \tag{10}$$

For this query, the counting method yields the program in Fig. 18:

$$\begin{aligned}
&cnt.anc(0, mark).\\
&cnt.anc(j+1, y) : -cnt.anc(j, z), P(y, z).\\
&anc'(j, x) : -cnt.anc(j, x).\\
&anc'(j-1, x) : -anc'(j, x).\\
&?anc'(0, x).
\end{aligned}$$

Fig. 18. Counting method for query (5.2) on rules of Fig. 15.

Observe now that the modified recursive rule performs no function other than decreasing the index by one, and copying the old values for x. But, since the initial value of j established by the exit rule is positive, j eventually becomes zero and x becomes an answer to the query. Therefore, we can dispense with the modified rules, and, hence, with the indices from the counting rules (which then look much as magic rules). The whole computation thus reduces to that of Fig. 19. Symmetric solutions work for the second formulation of the ancestor rule.

$$\begin{aligned}
&cnt.anc(mark)\\
&cnt.anc(y) : -cnt.anc(z), P(y, z).\\
&anc'(x) : -cnt.anc(x).\\
&?anc'(x).
\end{aligned}$$

Fig. 19. Specialized single Fixpoint program for query (5.2).

In summary, the counting method can be specialized to detect and eliminate a trivial first phase or trivial second phase, and produce a set of equivalent rules where (1) constants have been pushed into recursion, and (2) the constant arguments have been eliminated from the recursive predicates. Thus we have a solution that frees the user

Consider now the second query (5.2) on the same set of rules of Fig. 15.

$$?anc(x, mark) \tag{10}$$

For this query, the counting method yields the program in Fig. 18:

$cnt.anc(0, mark).$
$cnt.anc(j+1, y) : -cnt.anc(j, z), P(y, z).$
$anc'(j, x) : -cnt.anc(j, x).$
$anc'(j-1, x) : -anc'(j, x).$
$?anc'(0, x).$

Fig. 18. Counting method for query (5.2) on rules of Fig. 15.

Observe now that the modified recursive rule performs no function other than decreasing the index by one, and copying the old values for x. But, since the initial value of j established by the exit rule is positive, j eventually becomes zero and x becomes an answer to the query. Therefore, we can dispense with the modified rules, and, hence, with the indices from the counting rules (which then look much as magic rules). The whole computation thus reduces to that of Fig. 19. Symmetric solutions work for the second formulation of the ancestor rule.

$cnt.anc(mark)$
$cnt.anc(y) : -cnt.anc(z), P(y, z).$
$anc'(x) : -cnt.anc(x).$
$?anc'(x).$

Fig. 19. Specialized single Fixpoint program for query (5.2).

In summary, the counting method can be specialized to detect and eliminate a trivial first phase or trivial second phase, and produce a set of equivalent rules where (1) constants have been pushed into recursion, and (2) the constant arguments have been eliminated from the recursive predicates. Thus we have a solution that frees the user from having to customize the recursive rules to the particular pattern of bound/free argument in the goal, and also leads to a very efficient implementation.

While the domain of applicability of the techniques just proposed is limited, they supply good heuristics for cases such as transitive closures that are common in applications. In general, the question

The *query graph* $G_Q = \langle N,\ A\rangle$ is the subgraph of G, induced by all nodes that are reachable from a (recall that a denotes the constant or vector of constants in the query goal). We call a the *source node* of G_Q. The query graph G_Q, in turn, is composed by the three subgraphs $G_L = \langle N_L, A_L\rangle$, $G_E = \langle N_E, A_E\rangle$ and $G_R = \langle N_R, A_R\rangle$, such that A_L, A_E and A_R are all the arcs in A corresponding to pairs in L, E and R, respectively. It is easy to see that N_L and N_R are disjoint and contain L-nodes and R-nodes, respectively, and $N_L \cup N_R = N$. On the other hand, G_E is a bipartite graph having arcs from L-nodes to R-nodes. Finally, A_L, A_R and A_E are disjoint and $A_L \cup A_R \cup A_E = A$. The number of respective nodes of G_Q, G_L, G_R and G_E will be denoted by n, n_L, n_R and n_E, while the number of respective arcs is denoted by m, m_L, m_R and m_E.

Let b and c be two nodes in the query graph. If there is a path from b to c with length k, we say that c has a distance k from b.

Consider the graph G_L. It follows directly from the definitions that the nodes of this graph are the magic set values; $N_L = MS$. Thus we will call G_L the *Magic Graph* and refer to magic graph nodes and magic set values as synonyms. The counting set CS consists of pairs (j, b) where b is a node in the magic graph and j is its distance from the source node a, as it will be shown below. The set of values obtained from CS by projecting the indices out will be denoted CS_{-i}; obviously $CS_{-i} = MS$. Moreover, let I_b denote the set of indices j such that (j, b) is in CS. Then a node b will, respectively, be called

(a) *single* if I_b is a singleton set,
(b) *multiple* if I_b has a finite cardinality greater than one,
(c) *recurring* if I_b is infinite.

The magic graph of a query will be called *regular*, when all its nodes are single and non-regular otherwise.

For the query graph shown in Fig. 21, G_L is the subgraph induced by the nodes a, $a_1, \ldots, a_5$, while G_R is the subgraph, represented by darker arcs, induced by the nodes $b_1, \ldots, b_{10}$. The graph G_E is composed by the dashed arcs in Fig. 21. The magic graph is regular since all nodes in G_L are single, i.e., they have a unique distance from a. If we add the tuple $\langle a_2, a_5\rangle$ to the relation L then the query and the node a_5 become multiple; instead, if we add the tuple $\langle a_5, a_2\rangle$, then the query becomes cyclic and the nodes a_2, a_3 and a_5 become recurring.

Proposition 1. *Let Q be a query and G_Q be the query graph. Then $MS = CS_{-i} = N_L$. In addition, given a node b in G_L,*

a) b is single if and only if all directed paths from the source node a to b in G_L have the same distance,
b) b is multiple if and only if there at least two directed paths from a to b in G_L with different length,
c) b is recurring if and only if there is a cyclic directed path from a to b in G_L, and
d) the set I_b of indices associated with b coincides with the set of all distances of b from a.

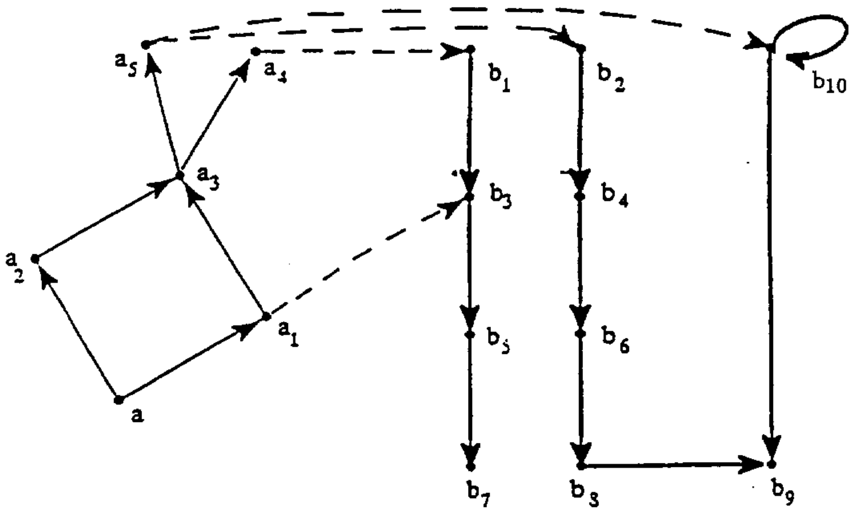

Fig. 21. Query Graph

As shown in Saccà and Zaniolo (1986) and Marchetti-Spaccamela et al. (1987), there is also a simple graph based interpretation of the query answer:

Fact 1. A node b is in the answer of Q if there is a (possibly cyclic) directed path from the source node a to b in G_Q such that this path is composed by exactly k arcs from A_L, one arc from A_E and k arcs from A_R, where k is any non-negative integer. □

Consider the query instance whose graph is shown in Fig. 21. Then b_5 is in the answer because of the path a, a_1, b_3, b_5. The other answer nodes are b_7, b_8, b_9 and b_{10}. Note that the latter node is in the answer because of the cyclic path

$$a, a_1, a_3, a_5, b_{10}, b_{10}, b_{10}, b_{10}. \tag{11}$$

The path from a to b_9 is cyclic as well.

Within this graph formalism, interesting complexity results about the magic set and the counting methods have been obtained in Marchetti-Spaccamela et al. (1987). The costs of the two methods are summarized in Table 1 for the different kinds of Magic Graphs (MG) The basic cost unit is the cost of retrieving a tuple in a database relation. We use the notations O and Θ for describing asymptotic time complexity. If the cost function of an algorithm is $f(n)$, where n is the problem size, and $g(n)$ is another function of n, then

a) $f(n) = O(g(n))$, if there exists a constant d such that $f(n) \leq d \times g(n)$ for all but some finite (possibly empty) set of non-negative values for n, and

b) $f(n) = \Theta(g(n))$, if both $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Table 1. Costs of the counting and magic set methods.

MG	Counting	MagicSet
Regular	$\Theta(m_L + n_L \times m_R)$	$\Theta(m_L \times m_R)$
Acyclic	$\Theta(n_L \times m_L + n_L \times m_R)$	$\Theta(m_L \times m_R)$
Cyclic	*unsafe*†	$\Theta(m_L \times m_R)$

† In Marchetti-Spaccamela et al. (1987) it has been shown that the counting method can be extended to deal with cyclic graphs and its cost is $\Theta(m \times n^2)$. Also note that the costs for the magic set method are actually higher than those given in Table 1, since arcs not in G_Q can be developed at each step of Fixpoint computation using the modified rules. For simplicity, these costs can be neglected since they simply reinforce the superiority of magic counting methods.

Proposition 2. *Let C and Ms be the costs of the counting method and of the magic set method, respectively. Then*

a) If the magic graphs are regular then $C = O(Ms)$.
b) If the magic graphs are acyclic and $m_L = O(m_R)$, then $C = O(Ms)$.

Proposition 2 says that the counting method always works better than the magic set method when the magic graphs are regular. In addition, since it is realistic to assume that m_R is, on the average, of the same order of m_L, it is fair to say that the counting method, on the average, works better than the magic set method when there is no cycle. In fact, in the average, we have that $C = \Theta(n \times m)$ and $Ms = \Theta(m \times m)$. Note that having $m_L \ll m_R$ is not sufficient for the magic set method to work better than the counting method.

Thus the counting method is superior to the magic set method in terms of worst case behavior. This superiority is even more dramatic when typical behavior is considered; in the comparative study presented in Brancilhon and Ramakrishnan (1986) the counting method was shown to be more efficient than all other methods by an order of magnitude (including the magic set method but excluding Prolog and the Henschen and Naqvi (1984) methods that often deliver comparable performance).

Unfortunately, the potential presence of cycles in the database compromises the applicability of the counting method in many situations. Note that, a database being logically acyclic (e.g., a non-incestuous family tree for the same generation example) does not guarantee that the physical database is cycle free, since checking acyclicity upon updates is very expensive and not often done in practice—thus there could be accidental cycles that throw the counting method astray. Therefore, a method that combines the performance of the counting set method with the safety of the magic set method is highly desirable.

7. Magic Counting Methods

We now propose a family of methods that combine the magic set and the counting methods and that are, therefore, called *magic counting* methods. All methods in the family make use of a *reduced magic set* and a *reduced counting set.* A *reduced* magic set, denoted by RM, is any (possibly empty and not necessarily proper) subset of the magic set MS. Likewise, RC will denote any (possibly empty and not necessarily proper) subset of the counting set CS, while RC_{-i} denotes the set of values in RC without their indices. In addition, for each b in RC_{-i}, RI_b is the set of all indices associated with b in RC (obviously $RI_b \subseteq I_b$).

The general structure of the magic counting methods consists of two steps. In the first step, a reduced magic set RM and a reduced counting set RC is constructed; in the second step, both the magic counting method and the magic set method are applied using the reduced sets. This second step is implemented as shown in Fig. 22.

Modified Rules & Query For MC Methods

$$S_C(J,\ Y)\ :-\ RC(J,\ X),\ E(X,\ Y). \quad (1)$$
$$S_C(J-1,\ Y)\ :-\ S_C(J,\ Y_1),\ R(Y,\ Y_1). \quad (2)$$

$$S_CM(X,\ Y)\ :-\ RM(X),\ E(X,\ Y). \quad (3)$$
$$S_M(X,\ Y)\ :-\ MS(X),\ L(X,\ X_1),\ S_M(X_1,\ Y_1),\ R(Y,\ Y_1). \quad (4)$$

$$Answer(X)\ :-\ S_C(0,\ X). \quad (5)$$
$$Answer(X)\ :-\ S_M(a, X). \quad (6)$$
$$?Answer(X)$$

Fig. 22. The independent magic counting method.

Observe that the modified rules of a magic counting method are basically the juxtaposition of the modified rules for the counting method (Rules 1 and 2) and those for the magic set method (Rules 3 and 4). However, they use reduced counting sets and magic sets. Thus, the predicate RC has replaced CS in the exit rule of S_C (Rule 1), while RM has replaced the original MS in the exit rule for S_M (Rule 3). Since Rules 1 and 2 operate independently from Rules 3 and 4, the methods conforming to the script of Fig. 22 are called independent magic counting methods. The following theorem characterizes the domain of correctness for these methods (Saccà and Zaniolo, 1987).

Theorem 1. *An independent magic counting method is correct when the following two conditions hold:*

a) $RM \cup RC_{-i} = MS$*, and*
b) for each b *in* $RC_{-i} - RM$, $RI_b = I_b$.

This theorem allows us to divide the nodes of the magic graph into the set RC that uses the counting method and the set RM that uses the magic set method. Since the counting method is better than the magic method for all nodes but the recurring ones, the ideal solution would assign the recurring nodes to RM and all others to RC. However, this ultimate goal is not easy to reach, since the existence of recurring nodes is not known at compile time and must be detected at run time at the price of some computational complexity. Thus, we present three alternative methods that approximate the ultimate goal with solutions that offer practical advantages of their own. Since detecting non-regular graphs is easier than detecting cyclic ones, these methods use the regularity of the magic graph as their decision criterion.

The simplest method to implement is the basic method, as follows:

a) *Basic Method.* If the graph G_L is regular then $RM = \emptyset$ and $RC = CS$, otherwise $RM = MS$ and $RC = \emptyset$. The basic method coincides with the counting method in the former case and with the magic set method in the latter case.

For instance, the graph G_L of Fig. 23 is not regular, thus $RM = MS = \{a, b, \ldots, l\}$ and $RC = \emptyset$.

While the basic method removes the compile-time dilemma of having to choose between counting and magic sets, it is clearly suboptimal since it does not, as it should, use the counting method for the parts of the graph which do not contain any multiple or recurring nodes. The next method accomplishes that by recording the level at which non-regular nodes are first found:

b) *Single Method.* Let k be the maximum index such that all nodes in CS_{-i} having an index less than k are single. Then, RC_{-i} is the set of all (single) nodes with index less than k, and $RM = MS - RC_{-i}$.

In Fig. 23, for example, we have $k = 2$, $RC_{-i} = \{a,\ b,\ c,\ d\}$ and $RM = \{e,\ f, \ldots,\ l\}$.

Using an index to partition the graph horizontally represents too coarse a criterion, since nodes in different vertical branches of the graph are smeared together. For the example of Fig. 23, for instance, the nodes e and f are assigned to RM, although they are single. The next method solves this problem:

c) *Multiple Method.* RC_{-i} is the set of all single nodes and $RM = MS - RC_{-i}$ (i.e., RM contains all multiple and recurring nodes).

For the example of Fig. 24, we have $RC_{-i} = \{a, b, c, d, e, f\}$ and $RM = \{g, h, i, j, k, l\}$.

Our final method uses counting for both single and multiple nodes.

d) *Recurring Method.* RC_{-i} is the set of all single and multiple nodes and $RM = MS - RC_{-i}$ (thus, RM contains all recurring nodes).

For the magic graph G_L in Fig. 24, the Recurring Method will produce, $RC_{-i} = \{a, b, c, d, e, f, h, k\}$ and $RM = \{g, i, j, l\}$.

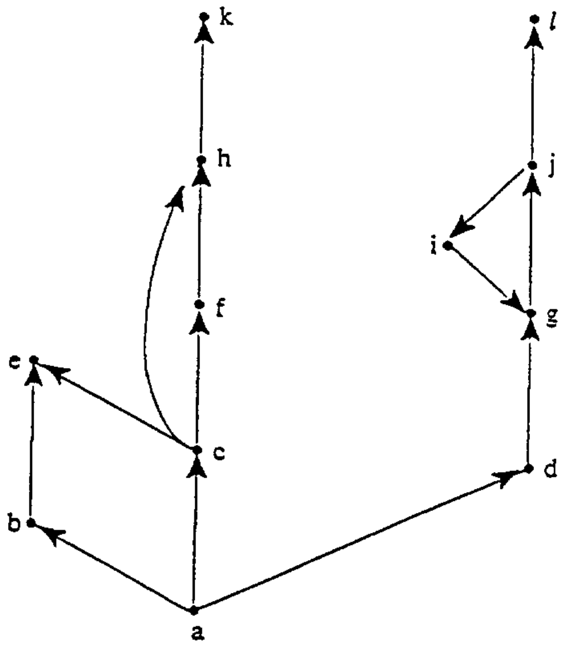

Fig. 23. Magic Graph

Before turning to the actual computation of the reduced set RC and RM, let us observe how, in the last three methods, the RM nodes have been relegated to the part of the graph most remote from the source—i.e., to the upper part of Fig. 24. As the magic set computation for these nodes progresses, it moves to the lower part of the graph (i.e, closer to the source node) where no recurring node exists—thus it can be improved by using the counting method. This idea has given birth to a further refinement of the magic counting method called an *integrated* magic counting method (Saccà and Zaniolo, 1987) (as opposed to the independent method just discussed). A discussion of the integrated counting method is outside the scope of this paper.

The computation of the sets RC and RM must be performed together, using a modification of the seminaive Fixpoint algorithm. The key idea is that the seminaive computation of the magic set and that of the counting set are very similar and, thus, can be combined together. Consider for instance the problem of implementing the Multiple Magic Counting method; here we must detect all single nodes and include them in the reduced counting set. To this end, we can use a temporary relation im, where the second column records whether we are dealing with the first occurrence of a node or the second one. Then, both first and second occurrences are used to generate other nodes in order to identify all multiple or recurring nodes (thus, we may need to use the same path twice) (see Fig. 24).

$$
\begin{aligned}
&\overline{MS}(0,1,a).\\
&\overline{MS}(j+1,1,X_1) :- \overline{MS}(j,1,X), L(X,X_1),\\
&\qquad \text{not } (\overline{MS}(I,\_,X_1), I \le j).\\
&\overline{MS}(j+1,2,X_1) :- \overline{MS}(j,\_,X), L(X,X_1), \overline{MS}(K,1,X_1), K \le j,\\
&\qquad \text{not } (\overline{MS}(I,2,X_1), I \le j).
\end{aligned}
$$

Fig. 24. Reduced sets computation for the multiple magic counting method.

At the end of the above Fixpoint computation, the multiple methods compute RC_{-i} as the set of all single nodes and RM as the set of all multiple/recurring nodes in the following way:

$$
\begin{aligned}
&MS(Y) :- \overline{MS}(\_,1,Y).\\
&RM(Y) :- \overline{MS}(\_,2,Y).\\
&RC(I,Y) :- \overline{MS}(I,1,Y), \text{ not } (RM(Y)).
\end{aligned}
\tag{12}
$$

If RC happens to be empty (i.e., there are no single nodes), then the integrated method adds the pair $(0, a)$.

The implementation of the recurring method is more complex, since the determination of cyclic structures in a graph is more suitable to depth-first search than to the breath-first expansion of the Fixpoint algorithm. An effective algorithm for detecting all recurring nodes consists in generating first the magic graph

$$ML(X, Y) : -MS(X), L(X, Y). \tag{13}$$

and then use the following algorithm to detect the non-recurring nodes $\overline{NR}(Y)$, with the help of an auxiliary predicate $UP(J, Y)$, which, at the end of each step J, contains all Y's that have an unsolved predecessor, i.e., there is an edge from some node in $\overline{NR}$ into Y.

$\overline{NR}(0, a) : -$ not $(ML(\_, a))$.
$UP(0, Y) : -ML(X, Y), X \neq a$.

$\overline{NR}(j + 1, Y) : -\overline{NR}(j, X), ML(X, Y),$ not $(UP(I, Y), I \leq J)$.
$UP(j + 1, Y) : -UP(j, Y),$ not $(\overline{NR}(j, Y))$.

Fig. 25. Computation of non-recurring nodes.

The program of Fig. 25 is locally stratified (Przymusinski, 1987); its perfect model can be computed efficiently using the Naive algorithm of Fig. 3 (Saccà and Zaniolo, 1988). At the end of this algorithm we have:

$$RC(X) : -\overline{NR}(\_, X). \tag{14}$$

Thus, all is left is to compute the indices for the set RC as shown in Fig. 26:

$RC(0, a)$.
$RC(j + 1, Y) : -RC(j, X), ML(X, Y), RC(Y)$.

Fig. 26. Computation of indices for non-recurring nodes.

Consider now the computational complexity of performing the seminaive computation on the rules of Fig. 25. Observe that for node Y is added to $\overline{NR}$ after n_m attempts, at most. Moreover, the size of UP is always less than n_L and the basic Fixpoint is performed m_m times. Thus, if n_m and m_m denote the number of non-recurring nodes and arcs between these, respectively, then the worst case complexity of computing ML and $\overline{NR}$ is $\Theta(m_L + n_m \times m_m + n_L \times m_m)$. (This upper bound is the actual one for complete graphs.) For the computation of Fig. 26, observe that for each node, there are at most m_m paths of different length between the node and the origin a. Thus, the complexity of the Fixpoint computation on the rules of Fig. 26 is $\Theta(n_m \times m_m)$. Thus the cost of computing RC is dominated by that of computing $\overline{NR}$.

It also easy to see how the computation of UP in Fig. 25 can be improved by deleting tuples from a UP table, rather than copying the entire table at each step. Then the complexity of the overall computation reduces to $\Theta(m_L + n_m \times m_m)$.

All magic counting methods are safe in the presence of cycles; moreover, they work better than the magic set method, and they coincide with the counting method when the query is regular (Saccà and Zaniolo, 1987). Furthermore, the recurring magic counting method behaves as the counting method, i.e., with complexity $\Theta(n_L \times m_L + n_L \times m_R)$, for acyclic graphs. In the integrated form discussed in Saccà and Zaniolo (1987), the recurring magic counting method has, for cyclic graphs, complexity $\Theta(m_L + n_m \times m_m) + (m_L - n_m) \times m_R + n_m \times m_R)$—thus it behaves as the counting method for all non-recurring nodes and as the magic set method for the recurring ones.

8. Conclusion

In this paper, we have presented a comprehensive solution that uses a Fixpoint based implementation for recursive predicates and compile-time rewriting techniques to make it safe and efficient. The first issue discussed was the application, via rule rewriting scripts, of symbolic finite differencing to improve the basic Fixpoint algorithm by removing redundant computation. Then, we tackled the key problem of taking advantage of bindings existing in recursive goals. To this end, the magic set method and the counting methods were developed which recast the original recursive rules into equivalent ones that are amenable to a safe and efficient implementation by two Fixpoint computations. A novel treatment was then proposed for dealing with those important cases, where one of these two fixed-points can be eliminated. A performance comparison between the magic set and the counting method was also presented; the results of this comparison motivated the introduction of a method, called the magic counting method, that integrates the strengths of the previous methods.

Many approaches were proposed in the past for supporting recursion in logic based languages; all these approaches have strengths and weaknesses (e.g., limited generality). The contribution of this paper consists in integrating some of the more powerful methods into a comprehensive framework which (i) provides a unified comparative treatment of the various methods, and (ii) supplies the basis for a general and robust system implementation where the various methods are chosen and combined to maximize their strengths and compensate for their weaknesses (Chimenti et al., 1988).

Acknowledgments

The authors are grateful to Catriel Beeri, Danette Chimenti, and Raghu Ramakrishnan for their comments and suggested improvements.

References

Aho, A. V., and Ullman, J. (1979). "Universality of Data Retrieval Languages," *Proc. POPL Conference*, San Antonio, Texas.

Balbin, I., and Ramamohanarao, K. (1987). "A Differential Approach to Query Optimization in Recursive Deductive Databases," *Journal of Logic Programming* **4**, No. 2, Sept. 1987, 259–262.

Bancilhon, F. (1985). "Naive Evaluation of Recursively defined Relations," *On Knowledge Base Management Systems* (M. Brodie and J. Mylopoulos, eds.), Springer-Verlag.

Bancilhon, F., and Ramakrishnan, R. (1986). "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. ACM SIGMOD Int. Conference on Management of Data*, Washington, D.C.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. (1986). "Magic sets and other strange ways to implement logic programs," *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems.*

Beeri, C., and Ramakrishnan, R. (1987). "On the Power of Magic," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems.*

Beeri, Nagvi, S., Ramakrishnan, R., Shmueli, O., and Tsur, S. (1987). "Sets and Negation in a Logic Data Language (LDL1)," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 269–283.

Beeri, C., Kanellakis,P., Bancilhon, F., and Ramakrishnan, R. (1987). "Bound on the Propagation of Selection into Logic Programs," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems.*

Chimenti D., O'Hare, T., Krishnamurthy, R., Nagvi, S., Tsur, S., Wert, C., and Zaniolo, C. (1988). "An Overview of the LDL System," *Database Engineering Bulletin.*

Chandra, A. K., and Harel, D. (1982). "Horn clauses and the Fixed-point hierarchy," *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 158–163.

Demo B., Porta, M., and Sapino, M. L. (1986). "A Stack Machine Semantics for Rewriting Rule Methods in Logical Databases." submitted for publication.

Gardarin, G., and De Maindreville, C. (1986). "Evaluation of Database Recursive Logic Programs as Recursive Function Series," *Proc. ACM SIGMOD Int. Conference on Management of Data*, Washington, D.C.

Gallaire, H., Minker, J., and Nicolas, J. M. (1984). "Logic and Databases: a Deductive Approach," *Computer Surveys* **16**, No. 2.

Henschen, L. J., and Naqvi, S. A. (1984). "On compiling queries in recursive first-order databases," *JACM 31*, 1, 47–85.

Kifer, M., and Lozinskii, E. L. (1986). "Filtering Data Flow in Deductive Databases," *ICDT '86*, Rome, Italy.

Kolaitis, G. P., and Papadimitriou, C. H. (1988). "Why Not Negation by Fixed-point?" *Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems.*

Lloyd, J. W. (1984). *Foundations of Logic Programming.* Springer-Verlag.

Morris, K. et al. (1987). "YAWN! (Yet Another Window on Nail!)," *Data Engineering* **10**, No. 4, 28–44.

Marchetti-Spaccamela, A., Pelaggi, A., and Saccà, D. (1987). "Worst-case complexity analysis of methods for logic query implementation," *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems.*

McKay, D., and Shapiro, S. (1981). "Using active connection graphs for reasoning with recursive rules," *Proc. 7th IJCAI*, 368–374.

Paige, R., and Koenig, S. (1982). "Finite Differencing of Computable Expressions," *ACM TOPLAS* **4**, No. 3, 402–454.

Przymusinski, T. (1987). "On the Semantics of Stratified Deductive Databases and Logic Programs." In *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), Morgan Kaufman, Los Altos.

Reiter, R. (1978). "On closed world databases." In *Logic and Databases* (H. Gallaire and J. Minker, eds.), Plenum, New York, 55–76.

Rohmer, J., Lescouer, R., and Kerisit, J. M. (1986). "The Alexander Method – A technique for the Processing of Recursive Axioms in Deductive Databases," *New Generation Computing* 4, No. 3, 273–287.

Saccà, D., and Zaniolo, C. (1986). "On the implementation of a simple class of logic queries for databases," *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems.*

Saccà, D., and Zaniolo, C. (1987). "Implementation of Recursive Queries for a Data Language based on Pure Horn Logic," *Proc. Fourth Int. Conference on Logic Programming*, Melbourne, Australia.

Saccà, D., and Zaniolo, C. "The Generalized Counting Method for Recursive Logic Queries," *JTC.* To appear. (Also *Proc. ICDT '86.*)

Saccà, D., and Zaniolo, C. (1987). "Magic Counting Methods," *ACM SIGMOD Proceedings.*

Saccà, D., and Zaniolo, C. "Differential Fixed-point Methods and Stratification of Logic Programs." Submitted for publication.

Tarski, A. (1955). "A Lattice Theoretical Fixed-point Theorem and its Application," *Pacific Journal of Mathematics No. 5,* 285–309.

Tsur, S., and Zaniolo, C. (1986). "LDL: A Logic-Based Data Language," *Proc. of 12th VLDB*, Tokyo, Japan.

Ullman, J. D. (1982). *Principles of Database Systems.* Computer Science Press, Rockville, Md.

Ullman, J. D. (1985). "Implementation of logical query languages for databases," *TODS* **10**, (3), 289–321.

van Emden, M. H., and Kowalski, R. (1976). "The semantics of Predicate Logic as a Programming Language," *JACM* **23** (4), 733–742.

Van Gelder, A. (1986). "A Message Passing Framework for Logical Query Evaluation," *Proc. ACM SIGMOD Int. Conference on Management of Data*, Washington, D.C.

Vieille, L. (1986). "Recursive Axioms in Deductive Databases: the Query-Subquery Approach," *Proc. First Int. Conference on Expert Database Systems*, Charleston, S.C.

Zaniolo, C. (1986). "Prolog: a database query language for all seasons." In *Expert Database Systems* (L. Kerschberg, ed.) Benjamin/Cummings.

Zaniolo, C. (1985). "The Representation and Deductive Retrieval of Complex Objects," *Proc. 11-th VLDB*, 459–469.

Zaniolo, C. "Safety and Compilation of Non-Recursive Horn Clauses," *Proc. First Int. Conference on Expert Database Systems.*

Zaniolo, C. (ed.), (1987). "Special Issue on Databases and Logic," *Data Engineering* **10**, No. 4.