Optimization of Linear Logic Programs Using Counting Methods

Sergio Greco*
Dip. di Elettr. Informatica Sistem.
Università della Calabria
87030 Rende, Italy
2101gre@icsuniv.bitnet

Carlo Zaniolo
Computer Science Department
University of California
LosAngeles, CA 90024
zaniolo@cs.edu.ucla

Abstract

We present a general solution to the problem of optimized execution of logic programs containing linear recursive rules. Our solution is based on extensions of the classical counting method, which is known to be efficient but of limited applicability. In fact, the range of applicability of the counting method, and its variants proposed by previous researchers, suffer from one or more of the following limitations: the method can be applied only when (1) the adorned program contains one recursive rule, (2) the 'left part' and the 'right part' of the recursive rule do not have any common variable and (3) the relation associated with the left part of the recursive rule is 'acyclic'. In this paper, a simple and unified framework is presented, where those limitations are removed, and the counting method thus become applicable to all programs with linear rules. This framework also allows a simple treatment of programs factorizable into segments, which can be computed separately yielding a much faster execution. A simple factorization technique based on argument reduction is presented that produces optimizations similar to those defined in the literature for RLC-linear programs (i.e., programs with right-linear rules, left-linear rules and a combination of the two).

1 Introduction

In this paper, we address the problem of evaluating queries with bound arguments on programs containing linear recursive rules. Several techniques, based on the propagation of bindings, have been defined in the literature to deal with this problem. These include the *magic-set* method [3,17,4], the *counting* method [3,17], and factorization techniques for special cases, such as those based on the reduction of the program [14] or on the combination of the propagation of bindings with the successive reduction [13].

While the magic-set method can be applied to all programs, the counting method and the factorization techniques can only be applied to restricted classes of linear programs—often yielding an order of magnitude of improvement in efficiency; comparisons between

*Work done while visiting MCC, Austin, Texas, and supported by the project "Sistemi Informatici e Calcolo Parallelo" obiettivo "Logidata+" of C.N.R. Italy.

the magic-set method and the counting method can be found in [4,11]. Many recursive programs found in practice are linear, and hence can be optimized by one of the specialized methods. Thus, the problem of identifying and processing these special situations is critical for an efficient implementation of deductive databases [20].

In this paper, we present a unifying framework for the efficient implementation of linear rules, as follows. We first propose extensions to the counting method, to make it applicable to all linear programs — even when the the database contains cycles. Furthermore, we show that, once these programs have been rewritten using the new counting method, the detection of special cases that are treatable by reduction techniques becomes simpler. Thus, we propose extensions to the optimizations defined in the literature [19,14] for programs containing left-linear and right-linear rules.

To illustrate the intuition underlying the different methods, we can use the following well-known example:

Example 1 Consider the following same-generation program *SG* with query sg(a, Y). The predicates up, down and flat are binary predicates that are non mutually recursive with sg.

```
sg(X,Y) <- flat(X,Y).
sg(X,Y) <- up(X,X1), sg(X1,Y1), down(Y1,Y).
```

The magic-set method first computes the set of tuples (called *magic set*) that are 'relevant' for the predicate sg and then uses the magic set to restrict the set of answers for the predicate sg. The magic-set method applies the modified goal sg(a, Y) on the following program:

```
m_sg(a).
m_sg(X1) <- m_sg(X), up(X,X1).
sg(X,Y) <- m_sg(X), flat(X,Y).
sg(X,Y) <- m_sg(X), up(X,X1), sg(X1,Y1), down(Y1,Y).
```

The counting method adds information on the distance from the initial node to each tuple belonging to the magic set, thus further constraining the later computation. In our example, the counting method produces the modified query, sg(a,Y,0), and the following program:

```
c_sg(a,0).
c_sg(X1,I+1) <- c_sg(X,I), up(X,X1).
sg(X,Y,I) <- c_sg(X,I), flat(X,Y).
sg(X,Y,I) <- c_sg(X,I), up(X,X1), sg(X1,Y1,I+1), down(Y1,Y).
```

This program can be further optimized, by dropping the first argument that is redundant for the example at hand, yielding query sg(Y,0), on the the following program:

```
c_sg(a,0).
c_sg(X1,I+1) <- c_sg(X,I), up(X,X1).
sg(Y,I) <- c_sg(X,I), flat(X,Y).
sg(Y,I) <- sg(Y1,I+1), down(Y1,Y).
```

The index I in c_sg(X, I) denotes the distance of the X from the source node a. The computation of the tuple of sg at level I uses only the tuples of sg computed at level I+1, i.e., the tuples computed in the previous step. In contrast, the magic-set method joins this set with the result of joining m_sg and up.

However, the basic counting method just described is not as general as the magic-set method, since it can be used only for a restricted class of linear programs and does deal well with cycles in base relations. Thus, in our example, if the relation up, is 'cyclic' the method generates a counting set with infinite tuples. Two different approaches have been defined for dealing with the problem of cycles. The first is based on the combination of the magic-set and the counting method [16], while the second extends the method for cyclic data [11,9,2]. The two methods, in any case, suffer of the other limitations of the classical method.

In this paper, we generalize the counting method to avoid the limitations of the current approaches. Moreover, our approach is conducive to further optimizations that are possible for more restrictive classes of (left-/right-) linear programs [14,13]. The paper is organized as follows. The next section introduces the basic definitions used in the paper. In section 3, an extension of the counting method to handle programs with acyclic databases is presented. Such a method is extended to programs with cyclic databases in section 4. In section 5, we present a technique for reduction of programs rewritten with the extended counting method. Due to space limitations we will omit the formal proofs of our theorems, which can be found in [8].

2 Basic definitions

We now recall some basic concepts [10,20]. Predicates whose definition consists of only ground facts are called *base* predicates while all others predicates are called *derived*. The set of facts whose predicate symbol is a base predicate defines the *database* while the set of clauses whose head predicates symbols is a derived predicate symbol defines the *program*. A *query* is a pair (G, P) where G is a query-goal and P is a program. The *answer* to a query (G, P) is the set of substitutions for the variables in G such that G is true with respect to the minimal model of P [10]. Two queries (Q, P) and (Q', P') are *equivalent* if they have the same answer for all possible databases.

Two variables X and Y in a rule r are connected if they appear in the same predicate or if there exists in r a variable Z such that X is connected to Z and Z is connected to Y. Two predicate P_1 and P_2 appearing in a rule are *connected* if there exists a variable appearing in both P_1 and P_2 or if there exist two connected variables appearing respectively in P_1 and P_2. A predicate p depends on a predicate q if 1) there exists a rule such that p appears in the head and q in the body or 2) if there exists a predicate s such that p depends on s and s depends on q. Two predicates p and q are mutually recursive if p depends on q and q depends on p.

We assume that the program has been partitioned according to a topological order $< P_1, ..., P_n >$. This means that each predicate appearing in P_i depends only on predicates belonging to P_j such that $j \leq i$. We assume also that the computation follows the topological order and that when we compute the component P_i the components $P_1, ..., P_{i-1}$ have been already computed. When we compute the component P_i all the facts obtained from the computation of the components $P_1,...,P_{i-1}$ are basically treated

the same as database facts. A rule in a component P_i is called *exit rule* if each predicate in the body belongs to a component P_j such that $j < i$. All the other rules are *recursive rules*. A recursive rule is said to be *linear* if the body of the rule contains at most one predicate that is mutually recursive with the head predicate. A program is linear if each rule is either an exit rule or a linear recursive rule. A linear recursive rule is of the form $P \leftarrow L, Q, R$ where P and Q are mutually recursive predicates while L and R are conjunctions of predicates not mutually recursive with P. We will denote the conjunctions L and R as *left* part and *right* part respectively.

An *adorned program* is a program whose predicate symbols have associated a string α, defined on the alphabet $\{b, f\}$, of length equal to the arity of the predicate. A character b (resp. f) in the i-th position of the adornment associated with a predicate p means that the i-th argument of p is bound (resp. free).

Let P be program and let P^c be the rewritten program obtained by applying the counting method to P. The program P^c contains a new set of predicates called *counting predicates*. The set of rules defining the counting predicates are called *counting rules*, while the remaining rules are called *modified rules*.

We assume that exit rules and recursive rules in an adorned program P^α have, respectively, the following form: $p(X,Y) \leftarrow E(B)$, and $p(X,Y) \leftarrow L(A)$, $q(X1,Y1)$, $R(B)$, where 1) the variable appearing inside the predicates denote lists of arguments; 2) p and q are mutually recursive predicates whose first and second arguments denote the lists of bound and free arguments, 3) E, L and R are (possibly null) conjunctions of predicates that are not mutually recursive with p and q, 4) the *safety* conditions $Y \subseteq (A \cup Y1 \cup B)$ and (obviously) $X1 \subseteq (X \cup A)$ hold. We assume also that the variables in the head are distinguished. There is no loss of generality in this assumption because each rule can be put in such a form by simple rewriting.

We now review the concept of query graph for an adorned program P [16,11]. Given a query $Q = (q(a,Y), P)$ and a database D we can associate to (Q, D) a graph called *query graph* defined as follows. An arc is a triplet (a, b, c) where are a and b are the source node and the destination node, while c is the label associated with the arc. Given an arc $e = (a, b, c)$ we say that the node a (resp. b) has in output (resp. input) the arc e.

The graph associated with (Q, D) is defined as follows:

1. there is an arc from x to $x1$ labeled (L, r, c) if there exists a ground rule r : $p(x,y) \leftarrow L(a), q(x1,y1), R(b)$ such that $L(a) \in D$ and c is the value for the variables appearing both in L and R;

2. there is an arc from $y1$ to y labeled (R, r, c) if there exists a ground rule r : $p(x,y) \leftarrow L(a), q(x1,y1), R(b)$ such that $r(b) \in D)$ and c is the value for the variables appearing both in L and R;

3. there is an arc from x to y marked (E, r) if there exists a ground rule $p(x,y) \leftarrow E(b)$ such that $E(b) \in D$.

The query graph G associated with a program can be partitioned into the three subgraphs G_L, G_R and G_E containing the arcs whose first argument of the label is L, R and E respectively.

Consider a graph representation G of a binary relation g. The set of arcs of g can be partitioned with respect to a node s (called source) into the following four disjoint classes [18,1]:

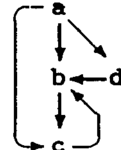
1. *Tree* arcs (g_t): these are the arcs defining the tree T_G obtained by the depth-first search visit of the graph starting from the node s;
2. *Forward* arcs (g_f): these are the arcs that go from a node v_1 to a node v_2 such that v_1 is an ancestor, but not a parent, of v_2 in T_G.
3. *Cross* arcs (g_c): these join two nodes that are not in the relation ancestor-descendant (in T_G).
4. *Back* arcs (g_b): these go from a node v_2 to a node v_1 such that v_1 is an ancestor of v_2 in T_G.

Tree arcs, forward arcs and cross arcs will be called *ahead* arcs. Notice that more than one different partitions are possible.

A node a is said to be *single* (resp. *multiple, recurring*) with respect to a source node s if there is a unique path (resp. a finite number greater than one, an infinite set of paths) from s to a. A graph is a tree if it contains only tree arcs (or equivalently, if each node is single), and is acyclic if it does not contain back arcs (or equivalently, if each node is non-recurring). A cycle is said to be *elementary* if each node is contained only once.

Example 2 Consider the graph defined by the relation *arc* and the source node a.

arc(a, b) arc(a, c) arc(d, b) arc(c, b)
arc(b, c)
arc(a, d)



Here, (a,b), (b,c) and (a,d) are tree arcs, (a,c) is a forward arc, (d,b) is a cross arc and (c,b) is a back arc. The nodes a and d are singles, while the nodes b and c are recurring. The arcs (b,c) and (c,b) define an elementary cycle. □

3 Acyclic Databases

For now, we leave the problem of cyclic databases to subsequent sections, and concentrate on the treatment of multiple recursive rules, and the situation where the left part and right part of a rule share variables. For this purpose, we replace the counting indexes with lists, which, operating as stacks, remember the state of the computation for later use. After proving the correctness of the method, we introduce a simpler implementation, where lists are replaced with pointers, to ensure the desired performance level for the method.

3.1 Multiple Linear Rules

The computation of a program rewritten by using the counting method (as well as each linear program rewritten by using the magic-set method) is executed in two different phases: the computation of the counting set and the computation of the answer. The method can thus be viewed as stack-based because during the first phase it remembers the number of applications of the (left part of the) recursive rule, and during the second

phase it executes (the right part of) the rule an equal number of times. The presence of more than one recursive rule implies that the exact sequence of rules used need to be memorized during the computation of the counting set, so that the same sequence of rules, but in reverse order, can be executed during the second phase. A list, having as entries the rule numbers, can be used for this purpose. We illustrate this point by means of an example.

Example 3 Consider the following program containing two recursive rules and the query-goal `sg(a,Y)`, which produces the adornment sg^{bf}.

$$
\begin{array}{lll}
r_0: & sg(X,Y) \leftarrow & flat(X,Y). \\
r_1: & sg(X,Y) \leftarrow & up1(X,X1),\ sg(X1,Y1),\ down1(Y1,Y). \\
r_2: & sg(X,Y) \leftarrow & up2(X,X1),\ sg(X1,Y1),\ down2(Y1,Y).
\end{array}
$$

The computation of the counting set associates to each element the sequence of rules applied to reach the element.

$$
\begin{array}{lll}
c_0: & c_sg^{bf}(a,[]). & \\
c_1: & c_sg^{bf}(X1,[r_1|L]) \leftarrow & c_sg^{bf}(X,L),\ up1(X,X1). \\
c_2: & c_sg^{bf}(X1,[r_2|L]) \leftarrow & c_sg^{bf}(X,L),\ up2(X,X1).
\end{array}
$$

If we have only one recursive rule, then all elements in the list are the same and it is sufficient to store the length of the list, as per the classical method. During the computation of the right part we need to use the reverse of the sequence of rules used to reach an element during the left part. For example, if we reached element x, starting from source node a, by the application of the left part of rules $r1, r1, r2, r1$, in the computation of the right part we need to apply the rules $r1, r2, r1, r1$. The rewritten rules, with the query-goal `sg(Y,[])`, are as follows:

$$
\begin{array}{lll}
r_0: & sg^{bf}(Y,L) \leftarrow & c_sg^{bf}(X,L),\ flat(X,Y). \\
r_1: & sg^{bf}(Y,L) \leftarrow & sg^{bf}(Y1,[r_1|L]),\ down1(Y1,Y). \\
r_1: & sg^{bf}(Y,L) \leftarrow & sg^{bf}(Y1,[r_2|L]),\ down2(Y1,Y).
\end{array}
$$

□

The use of lists could result in a performance overhead. In [15], a log of previously used rules was encoded into an integer. We will later propose a more efficient technique using pointers.

The transformation presented above is applicable to programs with more than one mutually recursive predicate and, consequently, to programs whose adornment in the recursive predicate in the body is different from the adornment in the head.

3.2 Shared Variables

The transformation presented in Section 3.1 applies only when the variables appearing in the right part of the recursive rules do not appear in the left part or among the bound variables in the head. This restriction permits us to partition the set of arguments of the recursive predicates into two distinct sets. If a variable in the right part of a recursive

rule also appears in the left part, or it is bound in the head, then we need to know its value when computing the answer. This implies that we need to store in the list the values of such variables. (Recall that the pair value and rule number is also labeling the arcs of the query graph associated with the program.) We show first the method by an example and in the next subsection we present the algorithm.

Example 4 Consider the the following example with the query-goal p(a, Y). In the second rule, the variable W appears both in the left and in the right part. In the third rule, the variable X is bound in the head and also appears in the right part.

r_0 : p(X, Y) ← flat(X, Z).
r_1 : p(X, Y) ← up1(X, X1, W), p(X1, Y1), down1(Y1, Y, W).
r_2 : p(X, Y) ← up2(X, X1), p(X1, Y1), down2(Y1, Y, X).

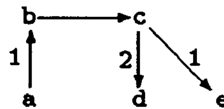
Each entry in the list defining the path contains two arguments: the identifier of the rule and a list containing the variables that are shared between the left and the right part of the rule. The values of the variables bound in the head appearing also in the right part of the rule (X in the third rule of our example) are not stored in the list because they appear also in the counting predicates. The resulting rewritten program (where the adornment bf have been omitted for brevity) is as follows:

c_p(a, [])
c_p(X1, [(r_1, [W])|L])) ← c_p(X, L), up1(X, X1, W).
c_p(X1, [(r_2, [])|L])) ← c_p(X, L), up2(X, X1).

p(Y, L) ← c_p(X, L), flat(X, Y).
p(Y, L) ← p(Y1, [(r1, [W])|L]), c_p(X, L), down1(Y1, Y, W).
p(Y, L) ← p(Y1, [(r2, [])|L]), c_p(X, L), down2(Y1, Y, X).

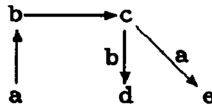
The goal c_p(X, L) in the second modified rule is not necessary since it is always true.

Consider the following database {up1(a, b, 1), flat(b, c), down1(c, d, 2), down1(c, e, 1)}, depicted as follows:



The original program computes the set S = {p(b, c), p(a, e)}. The set of facts computed by the rewritten program is {c_p(a, []), c_p(b, [(r_1, [1])]), p(c, [(r_1, [1])]), p(e, [])}. Moreover the rewritten program with the query p(Y, []) is equivalent to the original one with respect to the query p(a, Y).

Consider now the database {up2(a, b), flat(b, c), down1(c, d, b), down1(c, e, a)}, pictured in the following figure:



The rewritten program computes now the set {c_p(a, []), c_p(b, [(r_2, [])]), p(c, [(r_2, [])]), p(e, [])}. Also in this case the two program are equivalent with respect to the queries-goal p(Y, []) and p(a, Y). □

3.3 The Extended Counting Algorithm

For a given rule r, $L(r)$ and $R(r)$ denote, respectively, the left part and the right part of r, C_r denotes the list of variables appearing both in $L(r)$ and $R(r)$. D_r denotes the list of variables bound in the head appearing also in $R(r)$. There are three differences with respect to the classical method:

1. the index is substituted with a list whose elements are pairs (r, C) where r is the identifier of a rule and C is a value for C_r;
2. a counting predicate is added in the body of the recursive modified rules (necessary only if bound variable in the head appears also in the right part of the body);
3. the path argument in a counting (resp. modified) rules is incremented (resp. decremented) only if the program defining the predicate in the head of the original rule is not right- (resp. left-) linear.[1]

Algorithm 1 *[Extended Counting Rewriting]*
Input: Query $(q(a,Y), P)$ where the rules in P have form:
    Exit rules: $p(X,Y) \leftarrow e(B)$,
    Recursive rules: $p(X,Y) \leftarrow a(A), q(X1,Y1), b(B)$
Output: *rewritten query* $(q(Y,[]), P^{ec})$
begin
  % Generate Counting Rules
  $P^{ec} := \{c_q(a,[])\}$
  for each rec. rule r s.t. $(L(r)$ is not empty or $q \neq p$ or $X \neq X1)$ **do**
    if $R(r)$ is empty and $q = p$ and $Y = Y1$ **then**
      $P^{ec} := P^{ec} \cup \{c_q(X1,L) \leftarrow c_p(X,L), L(A)\}$
    else
      $P^{ec} := P^{ec} \cup \{c_q(X1,[(r,C_r)|L]) \leftarrow c_p(X,L), a(A)\}$
  % Generate Modified Rules
  for each exit rule **do**
    $P^{ec} := P^{ec} \cup \{p(Y,L) \leftarrow c_p(X,L), E(B\}$
  for each rec. rule r s.t. $R(r)$ is not empty or $q \neq p$ or $Y \neq Y1$ **do**
    if $L(r)$ is empty and $q = p$ and $X = X1$ **then**
      $P^{ec} := P^{ec} \cup \{p(Y,L) \leftarrow q(Y1,L), c_p(X,L), b(B)\}$
    else
      $P^{ec} := P^{ec} \cup \{p(Y,L) \leftarrow q(Y1,[(r,C_r)|L]), c_p(X,L), b(B)\}$
end.

Observe that the counting predicate $c_p(X,L)$ in the body of the modified recursive rules can be omitted if $D_r = \emptyset$, i.e., when no bound variable in the head also appear in the right part of the body.

Theorem 1 *Let $Q = (G,P)$ be an adorned query. Let Q' be the query obtained by application of algorithm 1 to Q. If the graph associated with (P,D) is acyclic, then Q and Q' are equivalent with respect to a database D.* □

[1] The definition of right and left linear programs are given in the next section

3.4 Implementation

The extended counting method adds to each counting predicate an argument denoting the path to reach the element starting from the initial binding. Such an argument, hereafter called *path argument*, is used to select the rule that must be used during the computation of the answer. The technique proposed in [15] (see also [6]) encodes such an information by using a number. Unfortunately this is not practical because the size of the number grows exponentially with the number of steps (the base of the number is equal to the number of rules in the rewritten program).

Our idea is to store for each element in the counting set only the rule used and the address of the tuple used to compute it. In particular, we assume that each tuple is associated with a unique identifier. We use an extended syntax to rewrite our programs. The extended syntax allows to use predicates of the form $O : P$, whose meaning is: "O is the identifier (in our case is the address) for the tuple P".[2] For example, if the element b in the counting set is computed by using the rule r and the tuple a we store the tuple $(b, r, [..], Addr(a))$.
Consider the program of Example 4. The set of counting rules is

```
c_p(a,r0,[],nil).
c_p(X1,r1,[W],A) ←   A : c_p(X,_,_,_),  up1(X,X1,W).
c_p(X1,r2,[],A) ←    A : c_p(X,_,_,_),  up2(X,X1).
```

The list associated with an element can be deduced by 'navigating' the chain defined by the last arguments. The set of rewritten rules is the following with the query-goal p(Y,_,nil) .

```
r0 : p(Y,R,A) ←  c_p(X,R,_,A),  e(X,Y).
r1 : p(Y,R,A) ←  p(Y1,r1,B),  B : c_p(X,R,[W],A),  down1(Y1,Y,W).
r2 : p(Y,R,A) ←  p(Y1,r2,B),  B : c_p(X,R,[],A),  down2(Y1,Y,X).
```

Notice that in this case when we compute the predicate $B : c_p(...)$ the variable B is bound and this corresponds to a direct access to the memory. Under this implementation, the method is very similar to the *Bushy-Depth-First* method used in the implementation of $\mathcal{LDL}$ [21].

If the graph associated with the left part of the recursive rules is acyclic and has n nodes, then in the worst case the counting set contains n^2 tuples. In comparison the set of tuples in the magic set is equal, in the worst case, to n. We propose next a modification of the method that reduces the size of the counting set to n. We show this by using the previous example. First we define the predicate up.

```
up(X,X1,[W],r1) ←  up1(X,X1,W).
up(X,X1,[],r2) ←   up2(X,X1).
```

A set of triplets is associated with each node in the counting set. Each element in the set contains the identifier of the rule, the set of common variables between the left and the right part of the rule and the identifier associated with the 'preceding node'. We use here the syntax of $\mathcal{LDL}$ that allows also set terms [5,12].[3] The program computing the counting set is:

[2] Many new logic languages support the concept of object ID. Here we use the notation of [22].
[3] $\mathcal{LDL}$ permits, by using grouping rules, to generate set terms.

$$\mathtt{c_p}(\mathtt{a}, \{(r_0, [], \mathtt{nil})\}).$$
$$\mathtt{c_p}(\mathtt{X1}, <(\mathtt{R}, \mathtt{V}, \mathtt{A})>) \leftarrow \mathtt{A} : \mathtt{c_p}(\mathtt{X}, _),\ \mathtt{up_a}(\mathtt{X}, \mathtt{X1}, \mathtt{V}, \mathtt{R}),\ \neg(\mathtt{up_a}(\mathtt{U}, \mathtt{X1}, _, _),\ \mathtt{U} \neq \mathtt{X},\ \neg(\mathtt{c_p}(\mathtt{U}, _)).$$

The predicate $\mathtt{up_a}$ denotes the set of tuples of up that are 'reachables' from the initial binding. Because we are assuming that the database is acyclic, $\mathtt{up_a}$ coincides with the set of ahead arcs in the query graph. Notice that such a program is not stratified. In the next section we show that it is 'weakly stratified' and that it can be computed efficiently.

4 Cyclic Databases

The counting method is unsafe if the database associated with the left part is cyclic, because it generates paths of infinite length. Two different techniques have been proposed for such a class of programs. The first, called *magic-counting* [16], combines the counting method and the magic-set method, while the second extends the counting method for programs with cyclic data [11,9,2].

We next present an extension of the counting method for cyclic database. Our method differs from those previously defined in the literature because such methods use specialized algorithms while our method is based on the simple rewriting of the original program. The idea is to associate to each vertex in G the acyclic distance from the source node and to nodes that have a back arc in input the length of the elementary cycles containing it.

Example 5 Consider the program of example 1. We, first, divide the set of up arcs that are reachables from the initial binding a (source node) into the two distinct subsets: (1) the ahead arcs $\mathtt{up_a}$ and (2) the back arcs $\mathtt{up_b}$. Such sets can be computed, in linear time, by using a variant of the depth first search algorithm [18,1]. We assume that the partition of the relation associated with the predicate up into the two subsets $\mathtt{up_a}$ and $\mathtt{up_b}$ has been already done. In this case the database associated with the predicate $\mathtt{up_a}$ is acyclic and the counting set can be computed as follows:

$$\mathtt{c_sg}(\mathtt{a}, \{(r_0, [], \mathtt{nil})\}).$$
$$\mathtt{c_sg}(\mathtt{X1}, <(r_1, [], \mathtt{A})>) \leftarrow \mathtt{A} : \mathtt{c_sg}(\mathtt{X}, _), \mathtt{up_a}(\mathtt{X}, \mathtt{X1}),\ \neg(\mathtt{up_a}(\mathtt{U}, \mathtt{X1}),\ \mathtt{U} \neq \mathtt{X},\ \neg(\mathtt{c_sg}(\mathtt{U}, _)).$$

After the computation of the predicate **c_sg**, a unique identifier is associated with each node in the counting set and such nodes are linked. We need now to add the links relative to the back arcs. The following predicate **cycle_sg** adds such information.

$$\mathtt{cycle_sg}(\mathtt{X1}, <(r_1, [], \mathtt{A})>) \leftarrow \mathtt{A} : \mathtt{sg_sg}(\mathtt{X}, _),\ \mathtt{up_b}(\mathtt{X}, \mathtt{X1}).$$

As shown in the previous section, our technique is to associate to each node in the right part of the graph a set of identifiers of the nodes in the left part and then to move in both parts using the same rule. Observe that it is also possible to move in the left part of the graph using back arcs. The predicate **f**, defined below, computes for a given identifier associated with a node in the counting set the set of identifiers of the nodes preceding it.

$$\texttt{f(A,S)} \leftarrow \texttt{A : c_sg(X,S1), if(cycle_sg(X,S2) then S} = \texttt{S1} \cup \texttt{S2 else S} = \texttt{S1)}.$$

The set of modified rules is then

$$\begin{array}{ll} r_0: & \texttt{sg(Y,S)} \leftarrow \texttt{A : c_sg(X,_), f(A,S), flat(X,Y)}. \\ r_1: & \texttt{sg(Y,S)} \leftarrow \texttt{sg(Y1,T)}, (r_1, [], A) \in T, \texttt{f(A,S), down(Y1,Y)}. \end{array}$$

The program contains only one recursive rule where no variables are shared between the left part and the right part of the rule. Thus, the arguments denoting the identifier of the rule and the list of the shared variables can be deleted. The resulting program, with the query-goal sg(Y, { nil }), is

$$\begin{array}{l} \texttt{c_sg(a,\{nil\})}. \\ \texttt{c_sg(X1,} < A >) \leftarrow \texttt{A : c_sg(X,_)}, \; up_a(X, X1), \\ \qquad \neg(up_a(W,X1), \; W \neq X, \; \neg c_sg(W,_)). \end{array}$$

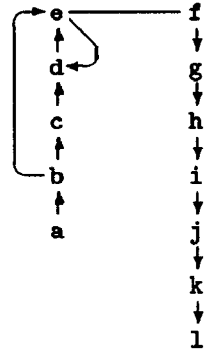
$$\texttt{cycle_sg(X1,} < A >) \leftarrow \texttt{A : c_sg(X,_)}, \; up_b(X,X1).$$

$$\texttt{f(A,S)} \leftarrow \texttt{A : c_sg(X,S1), if(cycle_sg(X,S2) then S} = \texttt{S1} \cup \texttt{S2 else S} = \texttt{S1)}.$$

$$\begin{array}{ll} r_0: & \texttt{sg(Y,S)} \leftarrow \texttt{A : c_sg(X,_), f(A,S), flat(X,Y)}. \\ r_1: & \texttt{sg(Y,S)} \leftarrow \texttt{sg(Y1,T)}, A \in T, \texttt{f(A,S), down(Y1,Y)}. \end{array}$$

Consider now the following database.

up(a,b)	down(f,g)	flat(e,f)
up(b,c)	down(g,h)	
up(c,d)	down(h,i)	
up(d,e)	down(i,j)	
up(e,f)	down(j,k)	
up(b,e)	down(k,l)	



The counting predicates associate to each value x for the variable X an identifier and the set of identifiers of elements preceding x through an ahead arc. The set of tuples for the predicate counting is then $\{ o_1 : (a, \{nil\}), o_2 : (b, \{o_1\}), o_3 : (c, \{o_2\}), o_4 : (d, \{o_3\}), o_5 : (e, \{o_2, o_4\}) \}$. The predicate cycle contains only the tuple $(d, \{o_5\})$.

The predicate f associates to each identifier of tuple in the counting set the identifiers of the tuples in the counting set 'preceding' it. The set of tuples for f is $\{(o_1, \{nil\}), (o_2, \{o_1\}), (o_3, \{o_2\}), (o_4, \{o_3, o_5\}), (o_5, \{o_2, o_4)\}$

Now se consider the computation of the tuples of sg. From the exit rule we obtain the tuple $(f, \{o_2, o_4\})$. Navigating on the relations down and c_sg from f and o_2 we obtain the tuple $(g, \{o_1\})$ while from f and o_4 we obtain $(g, \{o_3, o_5\})$. In the next step from g and o_1 we obtain the tuple $(h, \{nil\})$ that is an answer. From g and o_3, o_5 we obtain the tuples $(h, \{o_2\})$ and $(h, \{o_2, o_4\})$. The following steps compute the tuples $(i, \{o_1\})$, $(i, \{o_3, o_5\})$, $(j, \{nil\})$, $(j, \{o_2\})$, $(j, \{o_2, o_4\})$, $(k, \{o_1\})$, $(k, \{o_3, o_5\})$, $(l, \{nil\})$, $(l, \{o_2\})$, $(l, \{o_3, o_5\})$. □

We next present the algorithm for the extended counting method. We assume that each recursive rule r of the form $p(X,Y) \leftarrow a(A), q(X1,Y1), b(B)$, such that the conjunction denoted by the predicate a is not empty, has been replaced by the following two rules: $a'(X,X1,C_r,r) \leftarrow a(A)$ and $p(X,Y) \leftarrow a'(X,X1,C_r,r), q(X1,Y1), b(B)$. Notice that not all arguments in the predicate a' are necessary. We assume also that the relations associated with the predicate a' has been partitioned, with respect to the binding in the query-goal, into the two subset a'_a and a'_b denoting respectively the sets of ahead and back arcs.
Notation: $L(r)$ and $R(r)$ denote the left and the right part of a recursive rule r while C_r denote the set of common variables between $L(r)$ and $R(r)$.

Algorithm 2 *[Extended Counting Rewriting]*
Input: Query $(q(a,Y),P)$ as in Algorithm 1.
Output: rewritten query $(q(Y,\{(0,[],nil)\}),P^{ec})$
begin

% Generate Counting Rules
$P^{ec} := \{c_q(a,\{(r_0,[],nil)\})\}$
for each rec. rule r s.t. $L(r)$ in not empty or $q \neq p$ or $X \neq X1$ **do**
if $R(r)$ is empty and $q = p$ and $Y = Y1$ **then**
$P^{ec} := P^{ec} \cup \{c_q(X1, <(R,C_r,Id)>) \leftarrow c_p(X,T), (R,C_r,Id) \in T, a'_a(X,X1,_,_),$
$\neg(a'_a(W,X1,_,_), W \neq X, \neg c_p(W,_))\ \}$
else
$P^{ec} := P^{ec} \cup \{c_q(X1, <(R,C_r,Id)>) \leftarrow Id : c_p(X,_), a'_a(X,X1,C_r,R),$
$\neg(a'_a(W,X1,_,_), W \neq X, \neg c_p(W,_))\ \}$

% Generate Cycle Rules
for each rec. rule r s.t. $L(r)$ in not empty or $q \neq p$ or $X \neq X1$ **do**
if $R(r)$ is empty and $q = p$ and $Y = Y1$ **then**
$P^{ec} := P^{ec} \cup \{cycle_q(X1, <(R,C_r,Id)>) \leftarrow c_p(X,T),$
$(R,C_r,Id) \in T, a'_b(X,R,C_r,X1)\}$
else
$P^{ec} := P^{ec} \cup \{cycle_q(X1, <(R,C_r,Id)>) \leftarrow Id : c_p(X,_), a'_b(X,R,C_r,X1)\}$

% Generate Modified Rules
for each exit rule **do**
$P^{ec} := P^{ec} \cup \{\ p(Y,S) \leftarrow c_p(X,S), e(B,C_r),\ \ p(Y,S) \leftarrow cycle_p(X,S), e(B,C_r)\ \}$
for each rec. rule r s.t. $R(r)$ is not empty or $q \neq p$ or $Y \neq Y1$ **do**
if $L(r)$ is empty and $q = p$ and $X = X1$ **then**
$P^{ec} := P^{ec} \cup \{p(Y,T) \leftarrow q(Y1,T), (r,C_r,Id) \in T, Id : c_p(X,_), b(B,C_r)\}$
else
$P^{ec} := P^{ec} \cup \{p(Y,S) \leftarrow q(Y1,T), (r,C_r,Id) \in T, f(Id,S), Id : c_p(X,_), b(B,C_r)\}$

end.

Here, as in Algorithm 1, the counting predicate $Id : c_p(X,_)$ in the body of the modified recursive rules can be omitted if $D_r = \emptyset$, i.e., if no bound head variables appear in the right part of the body.

The following theorem elucidates the correctness of our extended counting method and the effective computability of the transformed program produced by the method.

Theorem 2 *Let $Q = (G,P)$ be an adorned query, and let $Q' = (G',P')$ be the query generated by Algorithm 2 applied to Q. Then, the following properties hold*

1. P' is weakly stratified.

2. Q and Q' are equivalent.

3. If P is a Datalog programs, the computation of the fixpoint of P' terminates. □

The rewritten program consists of a set of modified rules plus a set of *additional rules*. The set of modified rules depends on the set of additional rules. The set of additional rules, in the general case, contains the three distinct sets of rules: (i) counting rules, (ii) 'cycle' rules, and (iii) rules defining the predicate `f`. In order to compute the set of tuples for the additional rules we need to take the following steps: (1) partition the relation $L(r)$ into ahead-arcs and back-arcs; (2) compute the facts generated by counting rules; (3) compute the facts generated by the 'cycle' rules; (4) compute the predicate 'f'.

The computation of the additional rules, as described above, presents some inefficiencies. The partition of the relation associated with the left parts is done by using an algorithm that is a variation of the depth-first search on a graph. The nodes reached during the search coincides with the nodes in the counting sets. This means that the first two steps can be computed together. The information concerning the back arcs can be 'added' to the path argument of the tuples in the counting set. This means that also the predicate 'f' is not necessary. We assume that we use a *Bushy-Depth-First fixpoint* [7] that computes the set of additional rules that, with such assumption, consists only of counting rules. In practice the algorithm put the information given by the predicate 'f' into the tuples of the counting set. Under these assumptions, the predicate 'f' in the modified rules is no longer necessary, since the navigation is performed using directly the predicate counting that now contains information on back arcs.

Consider the program of Example 5. The set of tuples in the counting set is { o_1 : (a, {nil}), o_2 : (b, {o_1}), o_3 : (c, {o_2}), o_4 : (d, {o_3, o_5}), o_5 : (e, {o_2, o_4}) }.
The modified rules are:

```
r0 : sg(Y,S) ←  c_sg(X,S),  flat(X,Y).
r1 : sg(Y,S) ←  sg(Y1,T),  A ∈ T,  down(Y1,Y).
```

5 Reduction of Linear Programs

One of the advantages of the counting method is the ability to factorize the program, that is, to partition the recursive predicates and the rules defining them into two different recursive cliques. This implies that the new predicates and the rules defining them are simpler. For some classes of programs it is possible to factorize in a better way than the counting method do. This is the case of programs containing only 'left' and 'right linear' rules and only one recursive predicate. We show that the same optimization can be obtained by a simple reduction of the rewritten program. Such a reduction is based on the deletion of the path argument when it is not necessary.

A predicate is of the form $p(X, L)$ where L denotes the path argument and X denotes the list of the remaining arguments of p. Given a recursive rule r : $p(X, L_1) \leftarrow \ldots, q(Y, L_2), \ldots$ where q is the predicate mutually recursive with p, we say that r modifies the path argument of p if $L_1 \neq L_2$. Given a predicate p we say that the path argument of p is modified if and only if there exists a recursive rule that modifies it.

Algorithm 3 *[Program Reduction]*
input: A rewritten query Q^{ec}
output: A reduced query Q^{rec}
method: The reduced query Q^{rec} is obtained from the original program Q^{ec} by application of the following rules:
begin

1. *(deletion of argument)* the path argument of a set of mutually recursive predicates S can be deleted if for each predicate $p \in S$ no rules exist that modify the path argument of p;
2. *(deletion of predicates)* a counting predicate appearing in the body of a modified rule can be deleted if it is not connected with any predicate in the rule.

end.

Theorem 3 *Let $Q = (G, P)$ be an adorned query. Let Q' be the query obtained by application of Algorithm 2 to Q, and Q'' be the query obtained by application of Algorithm 3 to Q'. Q'',Q', and Q, are equivalent.* □

Next we show how programs that contain only left-linear and right-linear rules [19,14] can be reduced. Similar results are obtained in [13] by first applying the magic-set transformation and then factoring the rewritten program. Although their technique can be applied also to classes of non-linear programs, our technique, we believe, is simpler and can be extended to classes of non-linear programs.

A rule is said to be *right-linear* (*left-linear*) with respect to an adornment α if (1) the adornment of the recursive predicate in the body is α; (2) each variable in the head that is free (resp. bound) in α occurs in the same position in the recursive predicate in the body; (3) each free (resp. bound) variable in α occurs only once in the recursive predicate. A program is said to be *mixed linear* if it contains only right- and left-linear recursive rules and only one recursive predicate. A mixed linear program is said to be *right-linear* (resp. *left-linear*) if each rule is right- (resp. left-) linear.

When a program consists of left- and right-linear rules, the extended counting method and the reduction that follows can be combined. We show this by the use of an example.

Example 6

r_0 : p(X, Y) ← flat(X, Y).
r_1 : p(X, Y) ← up(X, X1), p(X1, Y).
r_2 : p(X, Y) ← p(X, Y), down(Y1, Y).

The rewritten program is

c_0 : c_p(a, []).
c_1 : c_p(X1, L) ← c_p(X, L), up(X, X1).
r_0 : p(Y, L) ← c_p(X, L), flat(X, Y).
r_3 : p(Y, L) ← p(Y1, L), down(Y1, Y).

The reduced program is

c_0 : c_p(a).
c_1 : c_p(X) ← c_p(X), up(X, X1).
r_0 : p(Y) ← c_p(X), flat(X, Y).
r_3 : p(Y) ← p(Y1), down(Y1, Y). □

Given a mixed-linear program P, we denote by $exit(P)$, $left(P)$ and $right(P)$ respectively, the sets of exit, left-linear and right-linear rules in P. Let P^{rec} be the reduced rewritten program P_c^{rec} and P_m^{rec} denote the sets of counting rules and the set of modified rules ($P^{rec} = P_c^{rec} \cup P_m^{rec}$). Given a program P, $\bar{P}$ denotes the program obtained from P by projecting out the path argument.

Fact 1 *Let P be a mixed-linear program.* $P^{rec} = exit(\bar{P}^{rec}) \cup left(\bar{P}_c^{rec}) \cup right(\bar{P}_m^{rec})$.

If the program contains only right-linear rules then $P^{rec} = \bar{P}_c^{rec} \cup exit(\bar{P}_m^{rec})$. For right-linear programs the reduction technique give the same optimized program presented in [14]. If the program contains only left-linear rules then $P^{rec} = exit(\bar{P}_c^{rec}) \cup \bar{P}_m^{rec}$. Observe that the rewritten program so obtained is similar to that of [14], except that, in the latter, there is no counting predicate, since the query bindings are pushed directly into the exit rule (in our example the exit rule is $p(Y) \leftarrow flat(a, Y)$).

6 Conclusion

In this paper, we introduced extensions of the counting method that makes it applicable to all programs with linear rules, and to the situations where the database contains cyclic data. The extended method is amenable to reduction techniques that produce further optimizations in the case of left-linear rules, right-linear rules and a combination of the two.

Furthermore, we presented a pointer-based implementation of the method that is computationally efficient. While more extensive measurements and evaluation are planned, our preliminary experience with the Bushy-Depth-First method in the $\mathcal{LDL}$ prototype [7] suggests that this approach yields excellent performance.

Aknowledgements
The authors are grateful to Mimmo Saccà for many useful discussions.

References

[1] A.V. Aho, Hopcroft J.E., and Ullman J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] H. Aly and Z.M. Ozsoyoglu. Synchronized counting method. In *Proceedings of the Fifth Intern. Conference on Data Engineering*, pages 366–373, 1989.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.

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

[5] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 21–37, 1987.

[6] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3 & 4):333–361, 1991.

[7] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, T. Shalom, and C. Zaniolo. The LDL system prototype. In *IEEE Transaction on Knowledge and Data Engineering*, pages 76–90, 1990.

[8] S. Greco and C. Zaniolo. *Optimization of Linear Logic Programs Using Counting Methods*. Research Report, MCC, 1991.

[9] R. Haddad and J. Naughton. A counting algorithm for a cyclic binary query. *Journal of Computer and System Science*, 43(1):145–169, 1991.

[10] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 2nd edition, 1987.

[11] A. Marchetti-Spaccamela, A. Pelaggi, and D. Saccà. Comparison of methods for logic query implementation. *Journal of Logic Programming*, 10(3 & 4):333–361, 1991.

[12] S. Naqvi and S Tsur. *A Logic Language for Data and Knowledge Bases*. Computer Science Press, New York, 1989.

[13] J.F. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.D. Ullman. Argument reduction by factoring. In *Proceedings of the 15th Conference on Very Large Data Bases*, pages 173–182, 1989.

[14] J.F. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the 1988 ACM SIGMOD Int. Conf. on Management of Data*, pages 235–242, 1989.

[15] D. Saccà and C. Zaniolo. The generalized counting method of recursive logic queries for databases. *Theoretical Computer Science*, 187–220, 1988.

[16] D. Saccà and C. Zaniolo. Magic counting methods. In *Proceedings of the 1987 ACM SIGMOD Int. Conf. on Management of Data*, pages 49–59, 1987.

[17] D. Saccà and C. Zaniolo. On the implementation of a simple class of logic queries for databases. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, pages 16–23, 1986.

[18] R Tarjan. Depth first search of linear graphs algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[19] J. Ullman. *Principles of Data and Knowledge-Base Systems*. Volume 2, Computer Science Press, New York, 1989.

[20] J. Ullman. *Principles of Data and Knowledge-Base Systems*. Volume 1, Computer Science Press, New York, 1988.

[21] C. Zaniolo. Design and implementation of a logic based language for data intensive applications. In *Proc. of the Intern. Conf. on Logic Programming*, 1988.

[22] C. Zaniolo. Object identity and inheritance in deductive databases: an evolotionary approach. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, 1989.