Efficient Execution of Recursive Queries Through Controlled Binding Propagation[*]

Sergio Greco
DEIS Dept.
Univ. of Calabria
87030 *Rende, Italy*
greco@ccusc1.unical.it

Carlo Zaniolo
Computer Science Dept.
Univ. of California
Los Angeles, CA 90024, *USA*
zaniolo@cs.ucla.edu

Abstract

This paper presents a new method for the computation of bound linear Datalog queries and compares its performance to that of other bottom-up execution strategies. The technique, called pushdown *method, uses virtual stacks to store information on the current state of the evaluation thus ensuring both termination and efficient execution. The asymptotic worst-case behavior and experimental results for the new method compare favorably against those of previous methods.*

1 Introduction

Several strategies has been proposed for the optimization of bound Datalog queries. Most of these strategies rewrite the original program into a query-equivalent new program which can be evaluated more efficiently by the semi-naive algorithm [2, 14, 4, 3, 9, 10, 12]. The improved efficiency of the rewritten program is due to the fact that it restricts the search to the portion of the underlying database that is relevant to the query. Other techniques present special algorithms that compute directly the original program [7, 17, 1, 6, 18].

Rewriting-based techniques can be classified according to their generality. Techniques that can be applied to all programs include the *magic-set* method and the *supplementary magic-set* method [2, 4]. Specialized techniques include the *factorization* and reduction of programs [9], the combination of the propagation of bindings with a successive reduction [10], and the *counting* method [2, 13, 1, 6]. These specialized techniques are important, since many programs of practical interest contain only linear recursive rules, to which these techniques apply, yielding an order of magnitude improvement in efficiency [16]. Comparisons between the magic-set method and the counting method can be found in [3, 8].

In this paper, we present a unifying framework for the efficient implementation of linear rules using a method, called *pushdown* method, which is based on the implicit use of stacks. For counting queries stacks reduce to counters and thus the method reduces to the classical counting method; for left-, right-, and mixed-linear queries the stacks can be deleted and the method reduces to the left-, right-, and mixed-linear method [5].

[*]The work of the first author has been supported by the CNR project “Sistemi Informatici e Calcolo Parallelo” and by the MURST project “Metodi Formali per Basi di Dati”.

2 Preliminaries

We now recall some basic concepts [16]. Predicates whose definition consists of only ground facts are called *base* predicates while all other 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 predicate symbols is a derived predicate symbol defines the *program*. A *query* is a pair (G, P) where G is a predicate called *query-goal* and P is a program. The *answer* to a query (G, P) on a database D is the set of substitutions for the variables in G such that G is true with respect to $(P \cup D)$. Two queries (G, P) and (G', 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 predicates P_1 and P_2 appearing in a rule are *connected* if they share some variable 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.

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 $j < i$; the other rules in the component are called *recursive rules*. A recursive rule is said to be *linear* if the body of the rule contains at most one predicate 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 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 call the conjunctions L and R as *left* part and *right* part respectively.

In an *adorned program*, occurrences of predicate symbols are adorned with superscript vectors: a b (resp. a f) in the i-th position of the adornment of a predicate p denotes that the i-th argument of p is bound (resp. free). Let P be a program, and P^c be the program obtained from P by applying a rewriting method, e.g., the magic-set method or the counting method. P^c contains a new set of predicates called, respectively, *magic* and *counting* predicates. The set of rules defining the magic (resp. counting) predicates are called *magic* (resp. *counting*) *rules*, while the remaining rules are called *modified rules*.

In general, exit rules and recursive rules in an adorned program P^α have, respectively, the following form

$$p(X,Y) \quad \leftarrow \quad \mathbf{e}(B) \tag{1}$$

$$p(X,Y) \quad \leftarrow \quad \mathbf{a}(A),\ q(X_1,Y_1),\ \mathbf{b}(B) \tag{2}$$

where 1) p and q are mutually recursive predicates whose first and second arguments denote the lists of bound and free arguments, 2) $\mathbf{a}, \mathbf{b}$ and $\mathbf{e}$ are (possibly empty) conjunctions of predicates that are not mutually recursive with p and q, 3) X, Y, X_1, Y_1, A, B denote lists of variables 4) the *safety* conditions $Y \subseteq (A \cup Y_1 \cup B)$ and $X_1 \subseteq (X \cup A)$ 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. The set of variables appearing in the right part of the rule which appear also in the left part or which are bound in the head ($(X \cup A) \cap B$) is the set of *shared variables*.

We now review the concept of query graph for an adorned program P [14, 8]. 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 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.

Let Q be an adorned query and let D be a database, the *query graph* associated with (Q, D) is defined as follows:

1. there is an arc from x to x_1 labeled $(left, r, c)$ if there exists a ground instantiation of an adorned rule $r : p(x, y) \leftarrow \mathbf{a}(a), q(x_1, y_1), \mathbf{b}(b)$ such that $\mathbf{a}(a) \subseteq D$ and c contains the values for the shared variables in r;
2. there is an arc from y_1 to y labeled $(right, r, c)$ if there exists a ground instantiation of an adorned rule $r : p(x, y) \leftarrow \mathbf{a}(a), q(x_1, y_1), \mathbf{b}(b)$ such that $r(b) \subseteq D$ and c is the value for the shared variables in r;
3. there is an arc from x to y marked $(exit, r)$ if there exists a ground adorned rule $r : p(x, y) \leftarrow \mathbf{e}(b)$ such that $\mathbf{e}(b) \subseteq D$.

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 *left*, *right* and *exit* respectively.

The *choice* construct of $\mathcal{LDL}$++ [19, 11], can be used to enforce functional constraints in rules. Thus, a goal of the form, `choice(X, Y)`, in a rule r denotes that any consequence derived from r must respect the FD $X \rightarrow Y$. In general, X can be a vector of variables — possibly an empty one denoted by " `( )`" — and Y is a vector of one or more variables. As shown in [15] the formal semantics of the construct can be given in terms of negation and stable model semantics.

3 Rewriting Datalog Queries

The computation of a program rewritten by using the counting method is executed in two phases: (i) the computation of the counting set and (ii) the computation of the answer. The method can thus be viewed as stack-based, since 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. If there is only one recursive rule, all the elements in the list are the same and then it is sufficient to store the length of the list, according to the classical method. The presence of more than one recursive rule means that the exact sequence of rules used in the first phase must be memorized, so that the same sequence of rules, but in reversed order, can be executed during the second phase. For example, if we reached element x, starting from source node a, by the application of the left part of rules r_1, r_1, r_2, r_1, in the computation of the right part we need to apply the rules r_1, r_2, r_1, r_1.

Furthermore, 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 modified rules. This implies that we need to store in the list the values of such variables. (Recall that the pair value–rule# is also labeling the arcs of the query graph associated with the program.) The following example shows how a linear program with shared variables and more than one recursive rule is rewritten using lists.

Example 1: *Consider the following example with the query-goal* $\mathtt{p(a,Y)}$*.*

$$\begin{array}{lll}
\mathtt{r_0}: & \mathtt{p(X,Y)} \leftarrow & \mathtt{flat(X,Z).} \\
\mathtt{r_1}: & \mathtt{p(X,Y)} \leftarrow & \mathtt{up1(X,X_1,W),\ p(X_1,Y_1),\ down1(Y_1,Y,W).} \\
\mathtt{r_2}: & \mathtt{p(X,Y)} \leftarrow & \mathtt{up2(X,X_1),\ p(X_1,Y_1),\ down2(Y_1,Y,X).}
\end{array}$$

The variable W in rule r_1 appears both in the left and in the right part while the variable X in rule r_3 is bound in the head and also appears in the right part. Each entry in the list defining the path contains two arguments: the identifier of the rule and a list containing the variables appearing in the right part appearing also in the left part or which are bound in the head. The resulting rewritten program (where the adornments have been omitted for brevity) is as follows:

$$\begin{array}{ll}
\mathtt{c\_p(a,[\ ])} & \\
\mathtt{c\_p(X_1,[(r_1,[W])|L]))} \leftarrow & \mathtt{c\_p(X,L),\ up1(X,X_1,W).} \\
\mathtt{c\_p(X_1,[(r_2,[X])|L]))} \leftarrow & \mathtt{c\_p(X,L),\ up2(X,X_1).}
\end{array}$$

$$\begin{array}{ll}
\mathtt{p(Y,L)} \leftarrow & \mathtt{c\_p(X,L),\ flat(X,Y).} \\
\mathtt{p(Y,L)} \leftarrow & \mathtt{p(Y_1,[(r1,[W])|L]),\ down1(Y_1,Y,W).} \\
\mathtt{p(Y,L)} \leftarrow & \mathtt{p(Y_1,[(r2,[X])|L]),\ down2(Y_1,Y,X).}
\end{array}$$

□

3.1 Implementation

The Pushdown method adds to each pushdown predicate an argument denoting the path connecting the initial binding to the element. Such an argument, from now on called *path argument*, is used to select the modified rule that must be used to compute the answer. For each element in the pushdown set we store the rule identifier, the list of shared variables and the address of the tuple used to compute it. In particular, we assume that each pushdown tuple is associated with an identifier, via the use of the $\mathcal{LDL}$++ choice construct. We use the notation $Id : P$ to show that "Id is the identifier for the tuple P" (Many new logic languages support the concept of *Object ID*.) Logically speaking, we can produce an equivalent program by simply making Id an additional argument of P; but the specialized notation is also suggestive of the implementation, since Id can be viewed as the address at which P is stored, and e.g., retrieving the values of attributes of P given Id is a unit cost operation.

For example, if the element b in the pushdown set is computed by using the rule r and the element a we store the tuple $(b, r, [..], Addr(a))$.

Example 2: Consider the program of Example 1. The pushdown rules are

$$\begin{array}{ll}
\mathtt{c\_p(a,r_0,[\ ],nil).} & \\
\mathtt{c\_p(X_1,r_1,[W],A)} \leftarrow & \mathtt{A : c\_p(X,\_,\_,\_),\ up1(X,X_1,W).} \\
\mathtt{c\_p(X_1,r_2,[X],A)} \leftarrow & \mathtt{A : c\_p(X,\_,\_,\_),\ up2(X,X_1).}
\end{array}$$

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 $\mathtt{A : c\_p(a, \_, \_, \_),\ p(Y, A)}$.

$$\begin{array}{lll} \mathtt{r_0:} & \mathtt{p(Y,A)} \leftarrow & \mathtt{A : c\_p(X, \_, \_, \_),\ e(X, Y).} \\ \mathtt{r_1:} & \mathtt{p(Y,A)} \leftarrow & \mathtt{p(Y_1, B),\ B : c\_p(\_, r_1, [W], A),\ down1(Y_1, Y, W).} \\ \mathtt{r_2:} & \mathtt{p(Y,A)} \leftarrow & \mathtt{p(Y_1, B),\ B : c\_p(\_, r_2, [X], A),\ down2(Y_1, Y, X).} \end{array}$$

□

Observe that here when we compute the predicate $B : c\_p(...)$ the variable B is bound and this corresponds to a direct access to the memory. Thus, the method is very similar to the *Bushy-Depth-First* method used in the implementation of $\mathcal{LDL}$ [19]. Notice also that the shared variables which appear also in the head could be omitted from the list of share variables since they appear also in the predicate pushdown appearing in the body of modified rules.

3.2 Cyclic Databases

The counting method is unsafe if the left part of the query graph is cyclic. Various techniques have been proposed to deal with these situations, including the *magic-counting* [14], that combines the counting method and the magic-set method, and more specialized algorithms [6, 8, 1]. Here, we will use the FD defined by choice goals in the rules, to avoid the generation of an infinite number of tuple identifiers. For instance, the rewritten program corresponding to the program of Example 3, is as follows:

$$\begin{array}{ll} \mathtt{B : c\_p(a, r_0, [\ ], nil).} \leftarrow & \mathtt{fd(a, B).} \\ \mathtt{B : c\_p(X_1, r_1, [W], A)} \leftarrow & \mathtt{A : c\_p(X, \_, \_, \_),\ up1(X, X_1, W),\ fd(X_1, B).} \\ \mathtt{B : c\_p(X_1, r_2, [X], A)} \leftarrow & \mathtt{A : c\_p(X, \_, \_, \_),\ up2(X, X_1),\ fd(X_1, B).} \end{array}$$

$$\begin{array}{lll} \mathtt{r_0:} & \mathtt{p(Y,A)} \leftarrow & \mathtt{A : c\_p(X, \_, \_, \_),\ e(X, Y).} \\ \mathtt{r_1:} & \mathtt{p(Y,A)} \leftarrow & \mathtt{p(Y_1, B),\ B : c\_p(\_, r_1, [W], A),\ down1(Y_1, Y, W).} \\ \mathtt{r_2:} & \mathtt{p(Y,A)} \leftarrow & \mathtt{p(Y_1, B),\ B : c\_p(\_, r_2, [X], A),\ down2(Y_1, Y, X).} \end{array}$$

where the rule defining fd is as follows:

$$fd(X, Y) \leftarrow choice((X), (Y))$$

The fd goals in the rules avoid the firing of rule instances that differ only in the values of the tuple identifiers.

3.3 The Linear Pushdown Algorithm

We next present the pushdown algorithm for linear programs.

Algorithm 1 *[Linear Pushdown Rewriting]*
Input: *Query* $(q(a, Y), P)$ as in Algorithm 1.
Input: *Adorned query* $(q(a, Y), P)$ where the rules have form

$p(X, Y) \leftarrow \mathbf{e}(B)$

$p(X, Y) \leftarrow \mathbf{a}(A), q(X_1, Y_1), \mathbf{b}(B)$

Output: *Rewritten query* $((A : c\_q(a, \_), q(Y, A)), P^{ec})$
Notation: $C_r = A \cap B$.

```
begin
    $P^{ec} := \{\ \}$
    % Generate Pushdown Rules
    $I : P^{ec} := P^{ec} \cup \{\ I : c\_p(a, (r_0, [\ ], nil))\ \} \leftarrow fd(a, I)\ \}$
    for each recursive rule $r$ s.t. $X \neq X_1$ do
      if $Y = Y_1$ and $p = q$ then
        $P^{ec} := P^{ec} \cup \{\ I : c\_p(X_1, (r, C_r, J)) \leftarrow c\_p(X, (r, C_r, J)),\ \mathbf{a}(A),\ fd(X_1, I)\ \}$
      else
        $P^{ec} := P^{ec} \cup \{\ I : c\_p(X_1, (r, C_r, J)) \leftarrow J : c\_p(X, \_),\ \mathbf{a}(A),\ fd(X_1, I).\ \}$
    % Generate Modified Rules
    for each exit rule do
      $P^{ec} := P^{ec} \cup \{\ p(Y, I) \leftarrow I : c\_p(X, \_), \mathbf{e}(X, Y),\ \}$
    for each recursive rule $r$ s.t. $Y \neq Y_1$ do
      if $X = X_1$ and $p = q$ then
        $P^{ec} := P^{ec} \cup \{p(Y, I) \leftarrow p(Y_1, I), I : c\_p(X, (r, C_r, J)),\ \mathbf{b}(B)\}$
      else
        $P^{ec} := P^{ec} \cup \{p(Y, I) \leftarrow p(Y_1, J), J : c\_p(\_, (r, C_r, I)),\ \mathbf{b}(B)\}$
end.
```

The predicate $I : c\_p(X, \_)$ in the body of the modified recursive rules can be omitted if no bound variable in the head 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. The computation of the fixpoint of the rewritten program Q' always terminates.* □

Theorem 2: *Let $Q = (G, P)$ be an adorned query. Let Q' be the query obtained by application of algorithm 1 to Q. Then Q and Q' are equivalent.* □

4 Complexity and Experimental Results

Let $Q = <G, P>$ be a query and let D be a database. The query graph $G = <V, E>$ associated with (Q, D) is composed of the three subgraphs $G_L = <V_L, E_L>$, $G_E = <V_E, E_E>$ and $G_R = <V_R, E_R>$ such that $V_L \cup V_R = V$ and $E_L \cup E_E \cup E_R = E$. Let $m_i(L)$ and $m_i(R)$ be the number of arcs with labels $(left, r_i, [..])$ and $(right, r_i, [..])$ respectively. Let $n(L)$ and $n(R)$ be the global number of nodes in G_L and G_R and let $m(L)$ and $m(R)$ be the global number of arcs in G_L and G_R, then

$$m(L) = \sum_{i=1}^{r} m_i(L) \qquad m(R) = \sum_{i=1}^{r} m_i(R)$$

where r is the number of recursive rules in P.
We assume that the costs of the operations are the following:

1. The cost to access a tuple $p(x, x1)$ is equal to the value of the cost access function $h(m)$ which depends on the access method and on the number of p-tuples m. In particular, the function, $h(m)$ is equal to (i) 1 when the address (tuple-identifier) of the tuple is known, (ii) a constant $c > 1$ when the key of the tuple is known, and (iii) $O(m)$ when the tuples are accessed sequentially. To distinguish access to base and derived relations we shall denote the cost access functions h_B and h_D respectively.

2. Given two sets S_1 and S_2, the costs for union and difference are linear in the size of the sets involved and are both equal to $h(m_1) \times m_2$ where $m_1 = min(|S_1|, |S_2|)$ and $m_2 = max(|S_1|, |S_2|)$. Moreover, if there are no indexes on the sets then we use sequential access and the cost for both union and difference is $O(m_1^2 \times m_2)$.

To compute the complexity we use the same-generation program

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

We assume that the number of tuples and constant in the base relations are comparable, i.e. $O(m(L)) = O(m(R)) = O(m)$ and $O(n(L)) = O(n(R)) = O(n)$.

Semi-naive Fixpoint: Since the recursive predicates have $O(n^2)$ tuples, the cost of adding a tuple is $O(h_D(n^2))$, while the cost of accessing a tuple from a base relation is $h_B(m)$. The computation corresponds to navigating both the left and the right parts of the query graph: thus, for each arc in the right part of the graph (accessed at cost $h_B(m)$) $O(m)$ arcs are selected from the left part, where the cost for each arc is also $h_B(m)$. Therefore the global cost is

$$O(m^2 \times (h_B^2(m) + h_D(n^2)).$$

Yannakakis' Method: The method proposed in [18] improves the computation of the fixpoint for chain programs, i.e., programs with binary predicates not containing shared variables. This method has complexity

$$O(m \times n \times h_D(n^2)).$$

In the present form, the method has very limited practical interest due to the fact that it does not make use of the bound query arguments (same as for the seminaive computation) and it is based on the construction of a ground program graphs of exceedingly large sizes.

We will next concentrate on methods that make effective use of the query constants. Therefore, let $\hat{m}$ and $\hat{n}$ respectively denote the number of tuples and constant in the database used to compute a bound query. That is, given a query $Q = <q(a, Y), P>$ and a database D, then $\hat{m} = \sigma(Q, D)$ and $\hat{n} = \sigma(Q, H)$ where σ is a *selection function* which takes in account the fact that the rewriting of the query allows us to use a restricted portion of the database ($0 < \sigma(Q, D) \leq m$ and $0 < \sigma(Q, H) \leq n$). Thus, although $O(\hat{m}) = O(m)$ and $O(\hat{n}) = O(n)$ we will use $\hat{m}$ and $\hat{n}$ when a restricted portion of the database is used.

The Magic-set method: The rewritten program consists of two sets of rules, called *magic* and *modified* rules respectively:

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

The number of tuples of the magic predicates (*m_sg* in our example) are bound by $O(\hat{n})$ while the number of tuples of the modified predicates (*sg* in our example) are bound by $O(\hat{n}^2)$. The computation of the magic rules takes cost $O(\hat{m} \times (h_B(m) + h_D(\hat{m}))$ since the number of body-tuples (satisfyed body conjunctions) is bound by $O(\hat{m})$ and each head-tuple must be added to a 'magic relation' with cost $O(h_D(\hat{m}))$. The factor $h_B(m)$ is the cost to access the tuples of the base relations (*up* in our example).

For the computation of the modified rules the only difference with respect to the semi-naive strategy is that we have an additional predicate in the body of the rules. The cost to produce a new tuple is now $h_B^2(m) \times h_D(\hat{m}) \times h_B^2(m)$, where $h_D(\hat{m})$ and $h_B(m)$ are the costs to access the tuples of the 'magic' and base relations. Therefore, the global cost is:

$$O(\hat{m}^2 \times (h_B^2(m) \times h_D(\hat{m}) + h_D(\hat{n}^2)).$$

The Supplementary Magic-set method: The method is similar to the magic-set one; however a supplementary relation is used in the modified rules, as follows:

$$\begin{array}{ll} \mathtt{sg(X,Y)} \leftarrow & \mathtt{s\_sg(X,X_1),\ sg(X_1,Y_1),\ down(Y_1,Y).} \\ \mathtt{s\_sg(X,X_1)} \leftarrow & \mathtt{m\_sg(X),\ up(X,X_1).} \end{array}$$

The size of the supplementary relation (*s_sg* in our example) is bound by $O(\hat{m})$ since it contains a subset of the tuples in the database (*s_sg* contains a subset of the tuples of *up*). The computation of the set of magic and supplementary rules has cost $O(\hat{m} \times (h_B(m) + h_D(\hat{m}))$.

For the computation of the modified rules we need to access the base relations of the right part of the rules (the relation *down* in our example) (at cost $h_B(m)$) and the supplementary relation (at cost $h_D(\hat{m})$). The global cost is then

$$O(\hat{m}^2 \times (h_B(m) \times h_D(\hat{m}) + h_D(\hat{n}^2))$$

The Acyclic Counting Method: The rewritten program consists of two sets of rules, called *counting* and *modified* rules, respectively, which for the example at hand define predicates *c_sg* and *sg* as follows:

$$\begin{array}{ll} \mathtt{c\_sg(a,0).} & \\ \mathtt{c\_sg(X_1,I+1)} \leftarrow & \mathtt{c\_sg(X,I),\ up(X,X_1).} \\ \mathtt{sg(Y,I)} \leftarrow & \mathtt{c\_sg(X,I),flat(X,Y).} \\ \mathtt{sg(Y,I-1)} \leftarrow & \mathtt{sg(Y_1,I),\ I>0,\ down(Y,Y_1).} \end{array}$$

The sizes of the counting and modified relations are both bound by $O(\hat{n}^2)$. The computation of the counting rules terminates in $O(\hat{n})$ steps and each iteration has cost $O(\hat{m} \times h_B(m))$ since $O(\hat{m})$ is the number of tuples used in the database and $h_B(m)$ is the cost to access a single tuple. The cost to add the new tuples is constant since these have an incremented index and we do not need to check for membership. The computation of the counting rules has then cost $O(\hat{n} \times \hat{m} \times h_B(m))$ and, thus, the computation of the modified rules has also cost $O(\hat{n} \times \hat{m} \times h_B(m))$. The global cost is then

$$O(\hat{n} \times \hat{m} \times h_B(m))$$

In passing, we also mention the extension proposed by [8] with complexity $O(n^2 \times m)$, that of [6] with complexity $O(n \times m)$, and that of [1] which has complexity $O(n^3)$. None of these proposals is based on some simple modification of the original algorithm; rather they use completely new algorithms.

	Range of applic.	Complexity
Semi-naive	General	$O(m^2 \times (h_B^2(m) + h_D(n^2))$
Yannakakis	Restricted Linear	$O(n \times m \times h_D(n^2))$
Magic-set	General	$O(\hat{m}^2 \times (h_B^2(m) \times h_D(\hat{m}) + h_D(\hat{n}^2))$
Suppl. magic	General	$O(\hat{m}^2 \times (h_B(m) \times h_D(\hat{m}) + h_D(\hat{n}^2))$
Counting	Restricted Linear	$O(\hat{m} \times \hat{n} \times h_B(m))$
Pushdown	All Linear	$O(\hat{m}^2 \times (h_B(m) + h_D(\hat{n}^2))$

Figure 1: *Asymptotic Complexity*

	m = 1000	m = 2000	m = 3000	m = 4000	m = 5000
Magic-set	407.33	547.39	671.30	791.14	891.66
Suppl. magic	21.64	33.29	43.39	53.06	61.29
Pushdown	7.03	13.40	18.90	24.06	28.34

Figure 2: *Experimental Results - sequential access -* $n = 200, \hat{n} = 50, \hat{m} = 100$

The Linear Pushdown Method: The rewritten program consists of two sets of recursive rules, called *pushing* and *modified* respectively, which for the example at hand define predicates *c_sg* and *sg* as follows:

```
B : c_sg(a, nil). ←   fd(a, B)
B : c_sg(X1, A) ←      A : c_sg(X, _),  up(X, X1),  fd(X1, B).

sg(Y, A) ←   A : c_sg(X, _),  e(X, Y).
sg(Y, A) ←   sg(Y1, B),  B : c_sg(_, A),  down(Y1, Y).
```

The number of tuples in the pushing relations is bound by $O(\hat{m})$, and the number of tuples in the modified relations is bound by $O(\hat{n}^2)$. The computation of the pushdown rules takes cost $O(\hat{m} \times h_B(m))$ because it is equivalent to the navigation of a graph with $\hat{m}$ arcs.

For the computation of the modified rules we need to access the base relations appearing in the right part of the rules (the relation *down* in our example) at cost $h_B(m)$ and the pushing relations (*c_sg* in our example) at cost 1. The cost to store a tuple is equal to $h_D(\hat{n}^2)$. The global cost is then

$$O(\hat{m}^2 \times (h_B(m) + h_D(\hat{n}^2))$$

The complexity results are resumed in the Table of Figure 1.

5 Conclusion

Due to space limitations, we can only give a short summary of the results of experiments discussed in [5]. These experiments confirm that the supplementary magic method is an order of magnitude better than the magic set method when sequential access is used. When extensible hashing is used for storing and retrieving tuples, then the latter only brings a modest 20% improvement with respect to the former. The pushdown method, however, is consistently four time faster than the supplementary magic method. The tables of Figure 2 and Figure 3 give typical results obtained on a PC with a 25-MH Intel 486 CPU.

	m = 1000	m = 2000	m = 3000	m = 4000	m = 5000
Magic-set	24.55	25.10	25.54	26.09	26.59
Suppl. magic	21.37	21.81	22.14	22.46	22.79
Pushdown	5.05	5.22	5.38	5.55	5.66

Figure 3: *Experimental Results - Extensible Hash* - $n = 200, \hat{n} = 50, \hat{m} = 200$

References

[1] H. Aly and Z.M. Ozsoyoglu. Synchronized counting method. In *Proc. of the Fifth Int. Conf. on Data engineering*, pp 366-373, 1989.

[2] F. Bancilhon, D. Mayer, Y. Sagiv, J.F. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. Fifth ACM PODS*, pp 1-15, 1986.

[3] F. Bancilhon and R. Ramakrisnhan. Performance evaluation of data intensive logic programs. *Found. of Ded. Dat. and L. Progr.*, pp 439-518, 1988.

[4] C. Beeri and R. Ramakrisnhan. On the power of magic. *Journal of Logic Programming*, 10 (3 & 4), pp 255-299, 1991.

[5] S. Greco and C. Zaniolo. Efficient Execution of Recursive Queries Through Controlled Binding Propagation. *Technical Report DEIS*, 1993.

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

[7] B. Lang. Datalog Automata. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases* Jerusalem, Israel, March, 1988, pp 389–401.

[8] A. Marchetti-Spaccamela, A. Pelaggi, D. Saccà. Comparison of methods for logic query implementation. *J. of Logic Programm.*, 10, pp 333-361, 1991.

[9] J. Naughton, R. Ramakrisnhan, Y. Sagiv, J.F. Ullman. Argument reduction by factoring. In *Proc. of the 15th VLDB Conference*, pp 173-182, 1989.

[10] J. Naughton, R. Ramakrisnhan, Y. Sagiv, J.F. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proc. of the SIGMOD Conf.*, 1989.

[11] R. Ramakrisnhan, D. Srivastava, S. Sudanshan. CORAL — Control, Relations and Logic. In *Proc. of 18th VLDB Conference*, 1992.

[12] R. Ramakrisnhan, Y. Sagiv, J. Ullman, M. Vardi. Logical Query Optimization by Proof-Tree Transformation. In *JCSS.*, No. 47, pp 222-248, 1993.

[13] D. Saccà and C. Zaniolo, The generalized counting method of recursive logic queries for databases. *Theor. Computer Science*, No. 62, 1988, pp 187-220.

[14] D. Saccà and C. Zaniolo, Magic-counting methods. In *Proc. of the 1987 ACM SIGMOD Int. Conf. on Management of Data*, pp 149-59, 1987.

[15] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. *Proc. of the Ninth ACM PODS*, pp 205–217, 1990.

[16] J.F. Ullmann. *Principles of Data and Knowledge-Base Systems.* Volule 1 & 2, Computer Science Press, New York, 1988.

[17] L. Vielle. Recursive Query processing: The Power of Logic. *Theoretical Computer Science*, 1989.

[18] M. Yannakakis. Graph-Theoretic Methods in Database Theory. In *Proc. of the Ninth ACM PODS*, 1990, pp 230–242.

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