The PushDown Method to Optimize Chain Logic Programs

(EXTENDED ABSTRACT)

Sergio Greco,[1] Domenico Saccà[1] and Carlo Zaniolo[2]

[1] DEIS, Univ. della Calabria, 87030 Rende, Italy
{ greco, sacca }@si.deis.unical.it
[2] Computer Science Dept., Univ. of California, Los Angeles, CA 90024
zaniolo@cs.ucla.edu

Abstract. The critical problem of finding efficient implementations for recursive queries with bound arguments offers many open challenges of practical and theoretical import. We propose a novel approach that solves this problem for chain queries, i.e., for queries where bindings are propagated from arguments in the head to arguments in the tail of the rules, in a chain-like fashion. The method, called *pushdown*, is based on the fact that a chain query can have associated a context-free language and a pushdown automaton recognizing this language can be emulated by rewriting the query as a particular factorized left-linear program. The proposed method generalizes and unifies previous techniques such as the 'counting' and 'right-, left-, mixed-linear' methods. It also succeeds in reducing many non-linear programs to query-equivalent linear ones.

1 Introduction

In the last decade bottom-up evaluation of logic programming has been favored by deductive database applications over traditional top-down approaches [18]. The effectiveness of the bottom-up execution for bound queries is based on optimizations techniques [4, 6, 10, 13, 14, 16, 18] that transform the original program into an equivalent one that efficiently exploits bindings during fixpoint-based computation. These rewriting techniques give the bottom-up computation a wider applicability range than the top-down computation typical of Prolog, and have been used successfully in several deductive database prototypes. However, there still remains room for major extensions and improvements.

In this paper we shall deal with chain queries, i.e., queries where bindings are propagated from arguments in the head to arguments in the tail of the rules, in a chain-like fashion [5, 8]. For these queries, insisting on general optimization methods (e.g., the *magic-set* method [18]) does not allow to take advantage of the chain structure, thus resulting in rather inefficient query executions. Therefore, as chain queries are rather frequent in practice (e.g., graph applications), there is a need for ad-hoc optimization methods. Indeed, various specialized methods for chain queries have been proposed in the literature (e.g., in [1, 5, 8, 22]). Unfortunately, these methods do not fully exploit possible bindings. To find a method that is particularly specialized for bound chain queries, we have to go back to the *counting* method; however, this method, although proposed in the context of general queries [17], preserves the original simplicity and efficiency [4, 18] only for a subset of chain queries whose recursive rules are linear.

In this paper, we present a new method for the optimization of bound chain queries that reduces to the counting method in all cases where the latter method behaves efficiently. Our approach is based on the fact that a chain query can be associated to a context-free language and a particular pushdown automaton recognizing this language can be also used to drive the query execution, thus dramatically reducing the complexity, as confirmed by the large number of experiments carried out in [7]. The so-called *pushdown* method translates a chain query into a factorized left-linear program implementing the pushdown automaton and, therefore, it candidates for a powerful rewriting technique for a large class of practical DATALOG programs.

Besides to giving an efficient execution scheme to bound chain queries and providing an extension of the counting method, another nice property of the new method is that it introduces a unified framework for the treatment of special cases, such as the factorization of right-, left-, mixed-linear linear programs, as well as the linearization of non-linear programs. A number of specialized techniques for the above special cases are known in the literature [13, 14, 23]. Given the importance and frequency of these special situations in practical applications, novel deductive systems call for the usage of a unique method that includes all advantages of the various specialized techniques.

We point out that analogies between chain queries and context-free languages were investigated by several authors, including [5, 2, 8, 9, 19, 20]. In particular, the use of automata to compute general logic queries was first proposed by Lang [11]. The Lang's method is based on pushing facts from the database onto the stack for later use in reverse order in the proof of a goal. As the method applies to general queries, it is not very effective for chain queries; besides, it does not exploit possible bindings. Independently in [21], Vielle proposed an extension of SLD-resolution which avoids replicated computations in the evaluation of general logic queries using stacks to perform a set-oriented computation. Also this method does not take advantage from a possible chain structure but it does exploit possible bindings. The first proposal of a method that is both specialized for chain queries and based on the properties of context-free language is due to Yannakakis [22], who has proposed a dynamic programming technique implementing the method of Cocke-Younger and Kasami to recognize strings of general context-free languages [3]. This technique turns out to be efficient for unbound queries but it does not support any mechanism to reduce the search space when bindings are available.

2 Preliminaries

We shall assume that the reader is familiar with basic definitions and concepts of logic programming [12] and of the DATALOG language [18]. We next present only definitions and notations that are specific to this paper.

A (*logic*) *program* is a set of *rules* that are negation free. The *definition of a predicate symbol* p in a program P, denoted by $def(p)$, is the set of rules having

p as head predicate symbol. A predicate symbol p is called *EDB* if all rules in $def(p)$ are facts (i.e., ground rules with empty body) or *IDB* otherwise.

Given two (not necessarily distinct) predicate symbols p and q, we say that $q \leq p$ if q occurs in the body of some rule in $def(p)$ or there exists a predicate symbol r such that $q \leq r$ and $r \leq p$; then $leq(p)$ denotes the set of predicate symbols q for which $q \leq p$. We say that p is *recursive* if $p \in leq(p)$ and that p and q are *mutually recursive* if $leq(p) = leq(q)$.

A rule with p as head predicate symbol is *recursive* if p is recursive, *linear* if it is recursive and there is exactly one predicate symbol in the body that is mutually recursive with p, *left-recursive* (resp., *right-recursive*) if the first (resp., the last) predicate symbol in the body is mutually recursive with p.

A *query* Q is a pair $<G, P>$ where G is an atom, called *query-goal*, and P is a program. The *answer* to the query Q, denoted by $A(Q)$, is the set of substitutions θ for the variables in G such that $G\theta$ is derived from P. Two queries $Q = <G, P>$ and $Q' = <G', P'>$ are *equivalent* if $A(Q) = A(Q')$.

Given a DATALOG (i.e., function-symbol free) program P and a set $\mathbf{q}$ of IDB predicate symbols occurring in P, a rule of P is a $\mathbf{q}$-*chain rule* if it has the following general format:

$$\begin{aligned} p_0(X_0, Y_n) \leftarrow \mathbf{a}_0(X_0, Y_0),\ p_1(Y_0, X_1),\ \mathbf{a}_1(X_1, Y_1),\ p_2(Y_1, X_2), ..., \\ \mathbf{a}_{n-1}(X_{n-1}, Y_{n-1}),\ p_{n-1}(Y_n, X_n),\ \mathbf{a}_n(X_n, Y_n). \end{aligned}$$

where $n \geq 0$, each X_i and Y_i, $0 \leq i \leq n$, are non-empty lists of distinct variables, each $\mathbf{a}_i(X_i, Y_i)$, $0 \leq i \leq n$, is a (possibly empty) conjunction of atoms whose predicate symbols neither are in $\mathbf{q}$ nor are mutually recursive with p_0, and each p_i, $1 \leq i \leq n$, is a (not necessarily distinct) predicate symbol in $\mathbf{q}$. We require that the lists of variables are pairwise disjoint; moreover, for each i, $0 \leq i \leq n$, if $\mathbf{a}_i(X_i, Y_i)$ is empty then $Y_i = X_i$ otherwise the variables occurring in the conjunction are all those in X_i and in Y_i plus possibly other variables that do not occur elsewhere in the rule.

If $n = 0$ —thus, r reduces to $p_0(X_0, Y_0) \leftarrow \mathbf{a}_0(X_0, Y_0)$.— then r is called an *exit chain rule*; moreover if $\mathbf{a}_0(X_0, Y_0)$ is the empty conjunction —thus r reduces to $p_0(X_0, X_0)$.— then r is called an *elementary chain* rule. Otherwise (i.e., $n > 0$), it is called a *recurrence chain* rule. Observe that a chain rule is linear iff it is recursive and $n = 1$; moreover, a chain rule is left-recursive or right-recursive iff $\mathbf{a}_0(X_0, Y_0)$ or $\mathbf{a}_n(X_n, Y_n)$, respectively, is the empty conjunction and p_1 or p_n, respectively, is mutually recursive with p_0.

A DATALOG program P is a $\mathbf{q}$-*chain program* if for each predicate symbol p in $\mathbf{q}$, every rule in $def(p)$ is $\mathbf{q}$-chain and for each two atoms $p(X, Y)$, $p(Z, W)$ occurring in the body or the head of $\mathbf{q}$-chain rules , $X = Z$ and $Y = W$ modulo renaming of the variables, thus the binding is passed through any atom of the same predicate symbol in $\mathbf{q}$ always using the same pattern.

A $\mathbf{q}$-*bound chain query* Q, is a query $<p(b, Y), P>$, where P is a $\mathbf{q}$-chain program, p is a predicate symbol in $\mathbf{q}$, and b is a bound argument.

In the next section we present a method which, given a $\mathbf{q}$-bound chain query $<p(b, Y), P>$, constructs an equivalent left-linear query. The program, so transformed can be implemented efficiently using the bottom-up least-fixpoint based computation favored by DATALOG [18]. In order to guarantee that the binding b is propagated through all $\mathbf{q}$-chain rules, we shall assume that $\mathbf{q} = \{p\} \cup \mathbf{q}'$, $\mathbf{q}' \subseteq leq(p)$ and for each q in $\mathbf{q}$, every $q' \in leq(p)$ for which $q \leq q'$ is in $\mathbf{q}$ as well. Moreover, in order to restrict optimization to those portions which depend from some recursion, we shall also assume that for each q in $\mathbf{q}$, there exists at least one recursive predicate symbol q' in $\mathbf{q}$ for which $q' \leq q$.

3 The Pushdown Method

Our method, called *pushdown method* is based on the analogy of chain queries and context-free grammars [19].

Example 1. Consider the simple chain query $Q = <\mathtt{sg}(\mathtt{b},\mathtt{Y}), P>$, on the following program P defining a non-linear same-generation program:

$\mathtt{sg}(\mathtt{X_0},\mathtt{Y_0}) \leftarrow \mathtt{a}(\mathtt{X_0},\mathtt{Y_0}).$
$\mathtt{sg}(\mathtt{X_0},\mathtt{Y_2}) \leftarrow \mathtt{b}(\mathtt{X_0},\mathtt{Y_0}),\ \mathtt{sg}(\mathtt{Y_0},\mathtt{X_1}),\ \mathtt{c}(\mathtt{X_1},\mathtt{Y_1}),\ \mathtt{sg}(\mathtt{Y_1},\mathtt{X_2}),\ \mathtt{d}(\mathtt{X_2},\mathtt{Y_2}).$

To this program there corresponds a context-free language generated by the grammar

$$G(Q) = <V_N, V_T, \Pi, sg>$$

where the set of non-terminal symbols V_N only includes the axiom sg, V_T is the set of terminal symbols $\{a, b, c, d\}$ and Π consists of the following production rules:

$sg \rightarrow a$
$sg \rightarrow b\ sg\ c\ sg\ d$

Note that the production rules in Π are obtained from the rules of P by dropping the arguments of the predicates and reversing the arrow.

The language $L(Q)$ generated by this grammar can be recognized by the automaton shown in Figure 1.

	b	c	d	a	ϵ
(q_0, Z_0)					$(q,\ sg\ Z_0)$
(q, sg)	$(q,\ sg\ c\ sg\ d)$			$(q,\ \epsilon)$	
(q, c)		$(q,\ \epsilon)$			
(q, d)			$(q,\ \epsilon)$		

Fig. 1. Pushdown Automaton for non-linear same generation query

This automaton can in turn be implemented by the following program $\hat{\Pi}$

$\mathtt{q}([\,\mathtt{sg}\,]).$
$\mathtt{q}([\,\mathtt{sg},\mathtt{c},\mathtt{sg},\mathtt{d}\,|\,\mathtt{T}\,]) \leftarrow \mathtt{q}([\,\mathtt{sg}\,|\,\mathtt{T}\,]),\ \mathtt{b}.$

$\mathtt{q}(\mathtt{T}) \leftarrow \mathtt{q}([\,\mathtt{sg}\,|\,\mathtt{T}\,]),\ \mathtt{a}.$
$\mathtt{q}(\mathtt{T}) \leftarrow \mathtt{q}([\,\mathtt{c}\,|\,\mathtt{T}\,]),\ \mathtt{c}.$
$\mathtt{q}(\mathtt{T}) \leftarrow \mathtt{q}([\,\mathtt{d}\,|\,\mathtt{T}\,]),\ \mathtt{d}.$

We can now construct a program $\hat{P}$ that is query-equivalent to P by reintroducing the variables in $\hat{\Pi}$. Thus, both X and Y variables are added to the non-recursive predicates. For the recursive predicate, we add the variable Y to the occurrences of the predicate in the head, and the variable X to the occurrences of the predicate in the body. The resulting program $\hat{P}$ is:

```
q(b, [sg]).                                   q(Y, T) ← q(X, [sg | T]), a(X, Y).
q(Y, [sg, c, sg, d | T]) ← q(X, [sg | T]), b(X, Y).   q(Y, T) ← q(X, [c | T]), c(X, Y).
                                              q(Y, T) ← q(X, [d | T]), d(X, Y).
```

It is easy to verify that the query $<\mathtt{q}(\mathtt{Y}, [\,]), \hat{P}>$ is equivalent to the original query. Observe that the rewritten program is not any-more DATALOG. □

In general, let us consider a **q**-chain query $Q = <p(b, Y), P>$. Let V be the set of all predicate symbols occurring in the **q**-chain rules; we have that **q** is the set V_N of non-terminal symbols and $V_T = V - V_N$. We associate to Q the context-free language $L(Q)$ on the alphabet V_T defined by the grammar $G(Q) = <V_N, V_T, \Pi, p>$ where the production rules in Π are as follows.

For each **q**-chain rule r_j of the form:

$$p_0^j(X_0, Y_n) \leftarrow \mathbf{a}_0^j(X_0, Y_0),\ p_1^j(Y_0, X_1),\ \mathbf{a}_1^j(X_1, Y_1),\ ...,\ p_n^j(Y_{n-1}, X_n),\ \mathbf{a}_n^j(X_n, Y_n)$$

with $n \geq 0$, there is the production rule:

$$p_0^j \rightarrow \mathbf{a}_0^j\ p_1^j\ \mathbf{a}_1^j \cdots \mathbf{a}_{n-1}^j\ p_n^j\ \mathbf{a}_n^j$$

The language $L(Q)$ is recognized by a two-state (q_0 and q, respectively initial and final state) pushdown automaton [15] whose transition table contains one column for each symbol in V_T plus a column for the ϵ symbol, one row for the pair (q_0, Z_0) and one row for each pair (q, v) where Z_0 is the starting pushdown symbol, and $v \in V$. (Note that, for the sake of presentation, the pushdown alphabet is not distinct from the language alphabet.) The Figure 2 reports the entry of the first row, corresponding to the start up of the pushdown consisting of entering the query goal symbol in the pushdown store, and the entries corresponding to the generic **q**-chain rule r_j shown above, one for $\mathbf{a}_0^j$ and one for each $\mathbf{a}_i^j$, $1 \leq i \leq n$, that is not empty. Obviously, if the rule is an exit rule (i.e., $n = 0$), the entry corresponding to $\mathbf{a}_0^j$ is (q, ϵ).

	$\mathbf{a}_0^j$	$\mathbf{a}_1^j$	$\cdots$	$\mathbf{a}_n^j$	ϵ
(q_0, Z_0)					$(q, p\,Z_0)$
$\cdots$					
(q, p_0^j)	$(q, p_1^j\mathbf{a}_1^j \cdots p_n^j\mathbf{a}_n^j)$				
(q, a_1^j)		(q, ϵ)			
$\cdots$					
(q, a_n^j)				(q, ϵ)	
$\cdots$					

Fig. 2. Pushdown Automaton recognizing $L(Q)$

Given a string $\alpha = a_{i_1}^{k_1} a_{i_2}^{k_2} \cdots a_{i_m}^{k_m}$ in V_T^*, a *path spelling α on P* is a sequence of $m + 1$ (not necessarily distinct) constants b_0, b_1, b_2, ..., b_m such that for each

j, $1 \leq j \leq m$, $a_{i_j}^{k_j}(b_{j-1}, b_j)$ is derived from P; if $m = 0$ then the path spells the empty string ϵ [1].

It is well known that c belongs to $A(Q)$ if and only if there exists a path from b to c, spelling a string α of $L(Q)$ on P. Therefore, in order to compute $A(Q)$, it is sufficient to use the automaton of Figure 2 to recognize all paths leaving from b and spelling a string α of $L(Q)$ on P [1]. This can be easily done by a logic program $\hat{P}$ which implements the automaton. The program $\hat{P}$ can be directly constructed using all transition rules of Figure 2. In particular we use a rule for each entry in the table. The start-up of the automaton is simulated by a fact which sets both the initial node of the path spelling a string of the language and the initial state of the pushdown store. For the chain query $Q = <p(b, Y), P>$, the resulting program, $\hat{P}$ is as follows:

$$\begin{array}{ll}
q(b, [p]). & \\
\ldots & \\
q(Y, [p_1^j, a_1^j, ..., p_n^j, a_n^j | T]) & \leftarrow q(X, [p_0^j | T]),\ \mathbf{a}_0^j(X, Y). \\
q(Y, T) & \leftarrow q(X, [a_1^j | T]),\ \mathbf{a}_1^j(X, Y). \\
\ldots & \\
q(Y, T) & \leftarrow q(X, [a_n^j | T]),\ \mathbf{a}_n^j(X, Y). \\
\ldots &
\end{array}$$

The rewritten program $\hat{P}$ will be called the *pushdown-program* of the query Q; the query $\hat{Q} = <q(Y, [\,]), \hat{P}>$ will be called the *pushdown-query* of Q. The technique for constructing pushdown-queries will be called the *pushdown method.*

Theorem 1. *Let Q be a* **q***-chain query. Then the pushdown-query of Q is equivalent to Q.* □

We point out that a naif execution of the rewritten program can be sometime inefficient or even non-terminating for cyclic databases. Extending the approach of [10] and as shown in [7], the bottom-up execution can be efficiently done by a suitable implementation of the list corresponding to the pushdown store. This list is represented as a pair consisting of the head and a pointer to the tuple storing the tail of the list. So, each possible cyclic sequence in the pushdown store is recorded only once and, therefore, termination is guaranteed. Moreover, the simulation of the method on a large number of examples in [7] has shown very better performances than traditional methods — for instance, the query of Example 1 is executed about five times faster if hash accesses are available or, even, one order of magnitude faster if only sequential accesses are enabled.

4 Grammar Transformations to improve Pushdown

As pointed out in the previous section, the pushdown method is based on constructing a particular pushdown automaton to recognize a context-free language. In this section we demonstrate that this kind of automaton becomes more effective if the grammar of the language has a particular structure. More interestingly, we show that if the grammar does not have this structure, then the program can be rewritten so that the corresponding grammar achieves the desired structure and this rewriting is mainly done by applying the known techniques for transforming grammars, particularly to get the $LL(1)$ format [3].

Let us first consider the case of a query for which every recursive chain rule is right-linear, i.e., both right-recursive and linear. Then the associated grammar $G(Q)$ is regular right-linear and, therefore, the pushdown can be replaced by a finite state automaton. In such cases, the pushdown method continues to work efficiently as the list implementing the pushdown store always contains at most one symbol; thus the pushdown automaton actually acts as a finite state automaton. An example is given next.

Example 2. Right-linear Transitive Closure. Consider the following right-linear **q**-chain query $Q =< \texttt{path}(\texttt{b},\texttt{Y}), P>$, where $\mathbf{q} = \{path\}$ and P is:

$\texttt{path}(\texttt{X},\texttt{Y}) \leftarrow \texttt{arc}(\texttt{X},\texttt{Y}).$
$\texttt{path}(\texttt{X},\texttt{Y}) \leftarrow \texttt{arc}(\texttt{X},\texttt{Z}),\ \ \texttt{path}(\texttt{Z},\texttt{Y}).$

The pushdown query is $<\texttt{q}(\texttt{Y},[\ \ ]), \hat{P}>$ where $\hat{P}$ is:

$\texttt{q}(\texttt{b},[\,\texttt{path}\,]).$
$\texttt{q}(\texttt{Y},\texttt{T}) \leftarrow \qquad\qquad \texttt{q}(\texttt{X},[\,\texttt{path}\,|\,\texttt{T}\,]),\ \ \texttt{arc}(\texttt{X},\texttt{Y}).$
$\texttt{q}(\texttt{Y},[\,\texttt{path}\,|\,\texttt{T}\,]) \leftarrow \texttt{q}(\texttt{X},[\,\texttt{path}\,|\,\texttt{T}\,]),\ \ \texttt{arc}(\texttt{X},\texttt{Y}).$

It is easy to verify that T is always empty, i.e., the bottom-up execution of the pushdown query emulates a finite state automaton and implements an efficient breadth-first search algorithm. □

Observe that if the grammar $G(Q)$ is regular left-linear then the pushdown method does not emulates a finite state automaton and, therefore, it may become rather inefficient or even non-terminating. As shown next, the problem can be removed by replacing left-recursion with right-recursion applying well-known reduction techniques for grammars [3].

Consider a **q**-chain query and suppose that a predicate symbol $s \in \mathbf{q}$ is in the head of some left-recursive chain rule — we call s *left-recursive* in this case. Then, the definition $def(s)$ consists of $m > 0$ left-recursive chain rules and n chain rules that are not left-recursive — obviously $n > 0$ as well because otherwise s would be trivially unsatisfied:

$$\begin{array}{ll} s(X,Y) \leftarrow \alpha_i(X,Y). & 1 \le i \le n \\ s(X,Y) \leftarrow s'(X,Z),\ \beta_i(Z,Y). & 1 \le i \le m \end{array}$$

The productions defining the symbol s in the grammar $G(Q)$ are:

$$\begin{array}{ll} s \to \alpha_i & 1 \le i \le n \\ s \to s',\ \beta_i & 1 \le i \le m \end{array}$$

where α_i and β_j denote the sequences of predicate symbols appearing in $\alpha_i(X,Y)$ and $\beta_j(Z,Y)$, respectively. We can then apply the known transformations to remove left-recursion from the second group of rules for all left-recursive predicate symbols s and we accordingly rewrite the corresponding rules. It turns out that the resulting program, denoted by $can(P)$, does not contain any left-recursive **q**-chain — here *can* stands for *canonical format.*

Example 3. Left-Linear Transitive Closure. Assume that the program P of the query of Example 2 is as follows:

$\texttt{path(X,Y)} \leftarrow \texttt{arc(X,Y)}.$
$\texttt{path(X,Y)} \leftarrow \texttt{path(X,U)},\ \ \texttt{arc(V,Y)}.$

The associated grammar $G(Q)$

$path \rightarrow arc \mid path\ arc$

is left recursive. After one step of the procedure for removing left-recursion, we obtain the right-recursive grammar

$path\ \ \rightarrow arc\ path'$
$path' \rightarrow arc\ path' \mid \epsilon$

So, the program $can(P)$:

$\texttt{path(X,Y)} \leftarrow\ \texttt{arc(X,Z)},\ \ \texttt{path}'\texttt{(Z,Y)}.$
$\texttt{path}'\texttt{(X,X)}$
$\texttt{path}'\texttt{(X,Y)} \leftarrow \texttt{arc(X,Z)},\ \ \texttt{path}'\texttt{(Z,Y)}.$

is right-linear as the program of Example 2. □

Example 4. Non-Linear Transitive Closure. Assume now that the program P of the query of Example 2 is defined as:

$\texttt{path(X,Y)} \leftarrow \texttt{arc(X,Y)}.$
$\texttt{path(X,Y)} \leftarrow \texttt{path(X,U)},\ \ \texttt{path(V,Y)}.$

This program is left recursive and, after the first step of the procedure for removing left-recursion, it is rewritten as:

$\texttt{r}_1 : \texttt{path(X,Y)} \leftarrow\ \texttt{arc(X,Z)},\ \ \texttt{path}'\texttt{(Z,Y)}.$
$\texttt{r}_2 : \texttt{path}'\texttt{(X,X)}.$
$\texttt{r}_3 : \texttt{path}'\texttt{(X,Y)} \leftarrow \texttt{path(X,Z)},\ \ \texttt{path}'\texttt{(Z,Y)}.$

The second step removes left recursion from the rule r_3 that is rewritten as

$$\texttt{r}_3 : \texttt{path}'\texttt{(X,Y)} \leftarrow \texttt{arc(X,W)},\ \ \texttt{path}'\texttt{(W,Z)},\ \ \texttt{path}'\texttt{(Z,Y)}.$$ □

Proposition 2. *Let $Q = <G, P>$ be a chain query and let $Q' = <G, can(P)>$. Then Q' is equivalent to Q.* □

We now introduce a program transformation that improves the performance of the pushdown method for an interesting case of right-recursion.

Let us suppose that there exists a predicate symbol s in P such that $def(s)$ consists of a single elementary chain rule — i.e., the rule $s(X, X)$.— and $m > 0$ right-recursive chain rules of the form:

$$s(X, Y) \leftarrow \alpha_i(X, Z),\ s(Z, Y). \qquad 0 \leq i \leq m$$

We rewrite each recursive chain rule that is in the following format:

$$s(X, Y) \leftarrow \alpha_i(X, Z),\ s(Z, W),\ s(W, Y).$$

as follows:

$$s(X,Y) \leftarrow \alpha_i(X,Z),\ s(Z,Y).$$

thus we drop one occurrence of the recursive goals at the end of the rule. Obviously, if the resulting rule has still multiple recursive goals at the end, we repeat the transformation. The program obtained after performing the above transformations for all the predicate symbols s in P is denoted by $simple(P)$.

Proposition 3. *Given a chain query $Q = <p(b,Y), P>$, Q is equivalent to $Q' = <p(b,Y),\ simple(P)>$.* □

Example 5. We have that $def(path') = \{r_2, r_3\}$ in the program $P' = can(P)$ of Example 4. The program $simple(P')$ is:

$\mathtt{r}_1 : \mathtt{path(X,Y)} \leftarrow\ \mathtt{arc(X,Z)},\ \mathtt{path'(Z,Y)}.$
$\mathtt{r}_2 : \mathtt{path'(X,X)}$
$\mathtt{r}_3 : \mathtt{path'(X,Y)} \leftarrow \mathtt{arc(X,U)},\ \mathtt{path'(U,Y)}.$

Eventually, we have linearized non-linear transitive closure. □

We observe that the transformation *simple* can be applied to a larger number of cases by applying further grammar rewriting. For instance, given the grammar:

$s\ \ \rightarrow a\, s'$
$s' \rightarrow b\, s\, s' \mid \epsilon$

we can modify it into:

$s\ \ \rightarrow a\, s'$
$s' \rightarrow b\, a\, s'\, s' \mid \epsilon$

so that we can eventually apply the transformation *simple*.

Example 6. Consider the {path}-chain query $Q = <\mathtt{path(b,Y)}, P>$ where P is defined as follows:

```
path(X,Y) ← yellow(X,Y).
path(X,Y) ← path(X,U), red(U,V), path(V,W), blue(W,Z), path(Z,Y).
```

We have that $can(P)$ is:

```
path(X,Y) ←  yellow(X,Z),  path′(Z,Y).
path′(X,X).
path′(X,Y) ← red(X,U),  path(U,W),  blue(W,Z),  path(Z,T),  path′(T,Y).
```

We now replace the two occurrence of `path` in the body of the last rule with the body of the first rule and we obtain the equivalent program P':

```
path(X,Y) ←  yellow(X,Z),  path′(Z,Y).
path′(X,X).
path′(X,Y) ← red(X,U),  yellow(U,V),  path′(V,W),  blue(W,Z),
             yellow(Z,T),  path′(T,S),  path′(S,Y).
```

By applying the transformation *simple* to `path′` the last rule of P' becomes:

```
path′(X,Y) ← red(X,U),  yellow(U,V),  path′(V,W),  blue(W,Z),
             yellow(Z,T),  path′(T,Y)
```
□

We now apply another transformation for the predicate symbols s for which the transformation *simple* cannot be applied because of the lack of the elementary chain rule. Let us then suppose that there exists a predicate symbol s in $\mathbf{q}$ such that $def(s)$ consists of $n > 0$ exit chain rules, say

$$s(X,Y) \leftarrow \beta_i(X,Y). \qquad 1 \le i \le n$$

and $m > 0$ right-recursive chain rules of the form:

$$s(X,Y) \leftarrow \alpha_i(X,Z),\ s(Z,Y). \qquad 1 \le i \le m$$

We rewrite the above rules as follows:

$$\begin{array}{lll} s(X,Y) & \leftarrow s'(X,Z),\ \beta_i(Z,Y) & 1 \le i \le n \\ s'(X,X). & & \\ s'(X,Y) & \leftarrow \alpha_i(X,Z),\ s'(Z,Y) & 1 \le i \le m \end{array}$$

We now replace possible atoms in α_i having s as predicate symbol with the bodies of the rules defining s. In this way, every rule will not have two consecutive recursive predicate symbols at the end of the body. The program obtained after performing the above transformations for all the predicate symbols s in P is denoted by $simple'(P)$. As confirmed by experiments in [7], the pushdown method becomes much more efficient when applied to $simple'(P)$ rather than to P.

Proposition 4. *Given a chain query $Q = <p(b,Y), P>$, Q is equivalent to $Q' = <p(b,Y),\ simple'(P)>$.* □

Example 7. Consider the query $Q = <\mathtt{path(b,Y)}, P>$ where P is as follows:

$$\begin{array}{l} \mathtt{path(X,Y)} \leftarrow \mathtt{yellow(X,Y)}. \\ \mathtt{path(X,Y)} \leftarrow \mathtt{red(X,V)},\ \mathtt{path(V,W)},\ \mathtt{path(W,Y)}. \end{array}$$

We obtain that $simple'(P)$ is equal to:

$$\begin{array}{l} \mathtt{path(X,Y)} \leftarrow\ \mathtt{path'(X,Z)},\ \mathtt{yellow(Z,Y)}. \\ \mathtt{path'(X,X)}. \\ \mathtt{path'(X,Y)} \leftarrow \mathtt{red(X,V)},\ \mathtt{path'(V,W)},\ \mathtt{yellow(W,T)},\ \mathtt{path'(T,Y)}. \end{array}$$

As discussed next, the format of $simple'(P)$ is very effective for the performance not only of the pushdown method but also of the counting method. □

5 When Pushdown reduces to Counting

In this section we describe some conditions under which the pushdown method reduces to the counting method. Actually, the counting method can be seen as a space-efficient implementation of the pushdown store. On the other hand, as the pushdown method has a larger application domain, we can conclude that the pushdown method is a powerful extension of the counting method.

Let us first observe that, given the pushdown program of a $\mathbf{q}$-chain query, the pushdown store can be efficiently implemented as follows whenever it contains strings of the form $\alpha^k(\beta)^n$, with $0 \le k \le 1$ and $n \ge 0$. Indeed the store can be replaced by the counter n and the introduction of two new states q_α and q_β

to record whether the top symbol is α or β, respectively. This situation arises when the program consists of a number of exit chain rules and of linear right-recursive chain rules and one single linear non-left recursive chain rule. Such an implementation corresponds to applying the counting method.

Example 8. Consider the linear program defining the same-generation with the query-goal $\mathtt{sg(d, Y)}$:

$\mathtt{sg(X, Y)} \leftarrow \mathtt{c(X, Y)}.$
$\mathtt{sg(X, Y)} \leftarrow \mathtt{a(X, X1),\ \ sg(X1, Y1),\ \ b(Y1, Y)}.$

The pushdown query is $<q(Y, [\,]), P'>$, where P' is:

$\mathtt{q(d, [\,sg\,])}.$
$\mathtt{q(Y, [\,sg, b\,|\,T\,])} \leftarrow \mathtt{q(X, [\,sg\,|\,T\,]),\ \ a(X, Y)}.$
$\mathtt{q(Y, T)} \leftarrow \qquad\quad \mathtt{q(X, [\,sg\,|\,T\,]),\ \ c(X, Y)}.$
$\mathtt{q(Y, T)} \leftarrow \qquad\quad \mathtt{q(Y, [\,b\,|\,T\,]),\ \ b(X, Y)}.$

Observe that the pushdown store contains strings of the form $sg(b)^n$ or of the form $(b)^n$, with $n \geq 0$. So, we replace the store with the counter n and the introduction of two new states q_{sg} and q_b to record whether the top symbol is sg or b, respectively. Therefore, the rules above can be rewritten in the following way:

$\mathtt{q_{sg}(d, 0)}.$
$\mathtt{q_{sg}(Y, I)} \leftarrow \mathtt{q_{sg}(X, J),\ \ a(X, Y),\ \ I = J + 1}.$
$\mathtt{q_{b}(Y, I)} \leftarrow\ \mathtt{q_{sg}(X, I),\ \ c(X, Y)}.$
$\mathtt{q_{b}(Y, I)} \leftarrow\ \mathtt{q_{b}(Y, J),\ \ b(X, Y),\ \ I = J - 1}.$

These rules are the same as those generated by the counting method. □

We now show that the above counting implementation of the pushdown store can be also done when the pushdown strings are of the form $\alpha^k\ (\beta\alpha)^n$ where $0 \leq k \leq 1$ and $n \geq 0$. This situation arises when the program consists of a number of exit chain rules and of recursive linear right-recursive chain rules and one single bi-linear (i.e., two recursive predicate symbols in the body) recursive chain rule that is right-recursive but not left-recursive, i.e., of the form:

$p(X_0, Y_2) \leftarrow \mathbf{a}_0(X_0, Y_0),\ p(Y_0, X_1),\ \mathbf{a}_1(X_1, Y_1),\ p(Y_1, Y_2)$

Example 9. Red/yellow path. Consider the query $<\mathtt{path(b, Y)}, P>$ where P is:

$\mathtt{path(X, X)}.$
$\mathtt{path(X, Y)} \leftarrow \mathtt{red(X, V),\ \ path(V, W),\ \ yellow(W, T),\ \ path(T, Y)}.$

Using the counting implementation of the pushdown store, we obtain

$\mathtt{q_{path}(b, 0)}$
$\mathtt{q_{yellow}(X, I)} \leftarrow \quad \mathtt{q_{path}(X, I)}.$
$\mathtt{q_{path}(Y, I + 1)} \leftarrow \mathtt{q_{path}(X, I),\ \ red(X, Y)}.$
$\mathtt{q_{path}(Y, I - 1)} \leftarrow \mathtt{q_{yellow}(X, I),\ \ yellow(X, Y)}.$

The query goal is $\mathtt{q_{yellow}(Y, 0)}$. Observe that the above program cannot be handled by the simple counting method as first introduced in [4]; the generalized counting method of [17] is able to implement the query but introducing additional indices. So we can say that, in such cases, the pushdown method is a simplified version of the counting method. □

Aknowledgements: Work partially supported by the EC-US project "Deus ex Machina". The work of the first two authors has been also supported by the projects "Sistemi Informatici e Calcolo Parallelo" of CNR and "Metodi Formali per Basi di Dati" of MURST.

References

1. F. Afrati, S. Cosmadakis. Expressiveness of Restricted Recursive Queries. In *Proc. ACM SIGACT Symp. on Theory of Computing*, 1989, pages 113–126.
2. F. Afrati, C.H.. Papadrimitriou. The parallel complexity of simple chain queries. In *Proceedings of the Sixth ACM PODS Conf.*, 1987, pages 210–213.
3. A.V. Aho, and J.F. Ullmann. *The Theory of Parsing Translating and Compiling.* Volume 1 & 2, Prentice-Hall, 1972.
4. F. Bancilhon, D. Mayer, Y. Sagiv, and J.F. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. Fifth ACM PODS*, 1986, pages 1–15.
5. C. Beeri, P. Kanellakis, F. Bancilhon, and R. Ramakrisnhan. Bounds on the Propagation of Selection into Logic Programs, *JCSS*, Vol. 41, No. 2, 1990, pages 157–180.
6. C. Beeri and R. Ramakrisnhan. On the power of magic. *Journal of Logic Programming*, 10 (3 & 4), 1991, pages 255–299.
7. F. Buccafurri, S. Greco, and E. Spadafora. Implementation of chain queries (in Italian) Proc. 2nd Italian Conference on Advance Database Systems. 1994.
8. G. Dong, On Datalog Linearization of Chain Queries. In J.D. Ullman, editor, *Theoretical Studies in Computer Science*, Academic Press, 1991, pages 181–206.
9. G. Dong, Datalog Expressiveness of Chain Queries: Grammar Tools and Characterization. In *Proc. Eleventh ACM PODS Conf.*, 1992, pages 81–90.
10. S. Greco and C. Zaniolo, Optimization of linear logic programs using counting methods. In *Proc. of the Extending Database Technology*, 1992, pages 187–220.
11. B. Lang. Datalog Automata. In *Third Conf. on Data and Knowledge Bases*, 1988.
12. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd ed., 1987.
13. J. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.F. Ullman. Argument Reduction by Factoring. In *Proc. 15th Conf. on Very Large data Bases*, 1989, pages 173–182.
14. J. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.F. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proc. SIGMOD Conf.*, 1989, pages 235–242.
15. J.E. Hopcroft, and J.F. Ullmann. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
16. R. Ramakrisnhan, Y. Sagiv, J.F. Ullman, and M.Y. Vardi. Logical Query Optimization by Proof-Tree Transformation. In *JCSS*, No. 47, pages 222-248, 1993.
17. D. Saccà and C. Zaniolo, The generalized counting method of recursive logic queries for databases. *Theoretical Computer Science*, Vol. 4, No. 4, 1988, pages 187–220.
18. J.D. Ullmann. *Principles of Data and Knowledge-Base Systems.* Comp. Sc., 1989.
19. J.D. Ullmann. The Interface Between Language Theory and Database Theory. In *Theoretical Studies in Computer Science* (J.D. Ullman, ed.), Academic Press, 1991.
20. J.D. Ullmann and A. Van Gelder. Parallel Complexity of Logical Query Programs. In *Proc. 27th IEEE Symp. on Found. of Computer Science*, 1986, pages 438–454.
21. L. Vielle. Recursive Query processing: The Power of Logic. *Theor. Comp. Sc.* 1989.
22. M. Yannakakis. Graph-Theoretic Methods in Database Theory. In *Proc. Ninth ACM Symposium on Principles of Database Systems*, 1990, pages 230–242.
23. W. Zang, C.T. Yu and D. Troy. Linearization of Nonlinear Recursive Rules. *IEEE Transaction on Software Engineering*, Vol. 15, No. 9, 1989, pages 1109–1119.