International Journal of Foundations of Computer Science
© World Scientific Publishing Company

GRAMMARS AND AUTOMATA TO OPTIMIZE CHAIN LOGIC QUERIES *

SERGIO GRECO
Dip. Elettronica Informatica e Sistemistica
Università della Calabria,
87030 Rende, Italy
greco@si.deis.unical.it

DOMENICO SACCÀ
Dip. Elettronica Informatica e Sistemistica
Università della Calabria,
87030 Rende, Italy
sacca@si.deis.unical.it

and

CARLO ZANIOLO
Computer Science Department
Univ. of California at Los Angeles
Los Angeles, CA, 90024
zaniolo@cs.ucla.edu

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The critical problem of finding efficient implementations for recursive queries with bound arguments offers many open challenges of practical and theoretical import. In particular, we need methods that are effective for the general case, such as non-linear programs, as well as for specialized cases, such as left-recursive linear programs. In this paper, 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 method*, is based on the fact that each chain query can be associated with a context-free language, and that 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 succeeds in reducing many non-linear programs to query-equivalent linear ones.

1. Introduction

1.1. Motivations

In the last decade, bottom-up evaluation of logic programs has been favored by deductive database applications over the traditional top-down approaches [31]. The effectiveness of the bottom-up execution for bound queries is based on optimizations techniques that transform the original program into an equivalent one that efficiently exploits bindings during fixpoint-based computation [5, 6, 8, 14, 22, 23, 26, 31]. 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. As discussed next, 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 rule, in a chain-like fashion [7, 9, 35]. For these queries, general optimization methods, such as the *magic-set* method [31], do not 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 specialized optimization methods. Indeed, several specialized methods for chain queries have been proposed in the literature (e.g., in [1, 7, 9, 11, 35, 36]). Unfortunately, these methods do not fully exploit the query bindings. On the other hand, the *counting* method is very effective for chain queries with bindings, for many chain queries whose recursive rules are linear; however, this method, although proposed in the context of general queries [29], looses its simplicity and efficiency [5, 31] for nonlinear queries, and even for some linear ones.

In this paper we propose a general method that exploits the relationships between chain queries and context-free languages. We shall show that classical grammar transformations can be applied to optimize our queries. Moreover, the well-known relationships between context-free languages and pushdown automata allows us to rewrite our queries into a form that are more suitable for a bottom-up evaluation.

In this paper, we shall use the deductive database language DATALOG [31] for the sake of simplicity and formal rigor of presentation. However, the techniques here presented can be used as well to optimize recursive SQL queries (e.g., SQL3 queries) [21].

1.2. Contributions

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 significantly reducing the complexity, as confirmed by the large number of experiments carried

out in [13, 12]. 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 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 [16, 17, 22, 23, 30, 35, 37]. 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.

1.3. Related work

The analogies between chain queries and context-free languages were investigated by several authors, including [7, 2, 9, 10, 25, 32, 33]. In particular, the use of automata to compute general logic queries was first proposed by Lang [18]. 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, 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 [34]. Also, this method does not take advantage of all possible chain structures but it does exploit possible bindings.

The first method that is specialized for chain queries and also based on the properties of context-free language is due to Yannakakis [36], who proposed a dynamic programming technique for implementing a procedure to recognize strings for general context-free languages that was originally due to Cocke-Younger and Kasami [3]. This technique turns out to be efficient for unbound queries, but does not support any mechanism to reduce the search space when bindings are available.

1.4. Plan of the paper

The remainder of this paper is organized as follows. In Section 2, we introduce the definition of chain programs and queries. In Section 3, we study their connections with context-free grammars and we present the pushdown method. In Section 4, we show that the analogy with context-free languages can be also exploited to use classical grammar transformations to rewrite the program in a format that is more suitable for the application of the pushdown method. Our analysis treats the class of left-recursive and right-recursive programs. In Section 5, we discuss the conditions under which the pushdown store can be replaced by a simple counter so that our method reduces to the counting method. In Section 6, we provide an implementation technique for the pushdown method. Finally, in Section 7, we present

some experimental results comparing our method with the classical magic-set and supplementary magic-set methods.

2. Preliminaries

We shall assume that the reader is familiar with basic definitions and concepts of logic programming [19] and of the DATALOG language [31]. 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 mutually recursive with some predicate symbol in the body, *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 $\langle G, P\rangle$ 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 $Q = \langle G, P\rangle$ and $Q' = \langle G', P'\rangle$ are *equivalent* if $A(Q) = A(Q')$.

Given a DATALOG (i.e., a 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:

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

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.

When $n = 0$, r reduces to $p_0(X_0, Y_0) \leftarrow \mathbf{a}_0(X_0, Y_0)$, and r is called an *exit chain rule*. Moreover, if $\mathbf{a}_0(X_0, Y_0)$ also reduces to the empty conjunction, r reduces to $p_0(X_0, X_0)$; then, r is called an *elementary chain* rule. Otherwise (i.e., when $n > 0$), r is called a *recurrence chain* rule. Observe that a chain rule is linear iff it is recursive and $n = 1$. A chain rule is left-recursive (resp. right-recursive) iff $\mathbf{a}_0(X_0, Y_0)$ (resp.

$\mathbf{a}_n(X_n, Y_n))$ is the empty conjunction and p_1 (resp. p_n) 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 $\langle p(b,Y), P\rangle$, where P is a $\mathbf{q}$-chain program, p is a predicate symbol in $\mathbf{q}$, b is a list bound arguments and Y is a list of variables.

In the next section, we present a method which, given a $\mathbf{q}$-bound chain query $\langle p(b,Y), P\rangle$, 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 [31]. 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 [32]. Without loss of generality, we can view our predicates as binary, by viewing the list of bound and unbound variables, as single bound/unbound arguments.

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

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

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

$$G(Q) = \langle V_N, V_T, \Pi, sg\rangle$$

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:

$$\begin{array}{lcl} sg & \rightarrow & a \\ sg & \rightarrow & b\ sg\ c\ sg\ d \end{array}$$

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

	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)			$(q,\ \epsilon)$		

Figure 1: *Pushdown Automaton for non-linear same generation query*

```
q([sg]).
q(T) ←                    q([sg|T]),  a.
q([sg,c,sg,d|T]) ←   q([sg|T]),  b.
q(T) ←                    q([c|T]),  c.
q(T) ←                    q([d|T]),  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 $\langle \mathtt{q(Y,[])}, \hat{P}\rangle$ is equivalent to the original query. Observe that the rewritten program is no longer pure DATALOG. □

In general, let us consider a **q**-chain query $Q = \langle p(b, Y), P\rangle$. 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 $G(Q) = \langle V_N, V_T, \Pi, p\rangle$. The production rules in Π are as follows:

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

$$p^j_0(X_0, Y_n) \leftarrow \mathbf{a}^j_0(X_0, Y_0),\ p^j_1(Y_0, X_1),\ \mathbf{a}^j_1(X_1, Y_1),\ ...,\ p^j_n(Y_{n-1}, X_n),\ \mathbf{a}^j_n(X_n, Y_n)$$

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

$$p^j_0 \quad \rightarrow \quad \mathbf{a}^j_0\ p^j_1\ \mathbf{a}^j_1 \cdots \mathbf{a}^j_{n-1}\ p^j_n\ \mathbf{a}^j_n$$

The language $L(Q)$ is recognized by a two-state (q_0 and q, respectively initial and final state) pushdown automaton [24] whose transition table contains one column for each symbol in V_T, plus a column for the symbol ϵ. The transition table has one row for the pair (q_0, Z_0), where Z_0 is the starting pushdown symbol, and one row for each pair (q, v) with $v \in V$. (Note that, for the sake of presentation,

	$\mathbf{a}_0^j$	$\mathbf{a}_1^j$	$\cdots$	$\mathbf{a}_n^j$	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)	
$\cdots$					

Figure 2: *Pushdown Automaton recognizing $L(Q)$*

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).

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 ing 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 $Q = \langle p(b, Y), P \rangle$, the resulting program, $\hat{P}$ is as follows:

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

The rewritten program $\hat{P}$ will be called the *pushdown-program* of the query Q; the query $\hat{Q} = \langle q(Y, [\,]), \hat{P} \rangle$ 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.*

Proof. Let $Q = \langle p(b, Y), P \rangle$ be a **q**-chain query and $\hat{Q}$ the pushdown query of

Q. Recall that $G(Q)$ is the grammar associated with P and $L(Q)$ is the language generated by $G(Q)$. A constant c belongs to the answer set $A(Q)$ if and only if there exists a path from b to c spelling a string of $L(Q)$ on P. Let $PD(Q)$ be the automaton associated with the grammar $G(Q)$. It is well known that a string generated by any grammar G is recognized by empty store by the pushdown automata, whose transitions are of the form

$$\begin{array}{rcl}\delta_G(q, a, K) & = & \{(q, \alpha) \mid K \rightarrow a\,\alpha\} \in G \\ \delta_G(q, a, a) & = & \{(q, \epsilon)\}\end{array}$$

where a is terminal symbol and K is non terminal [24]. It is easy to see that this automaton for $G(Q)$ is $PD(Q)$. Hence, the set of strings generated by $G(Q)$ coincides with the set of strings recognized by $PD(Q)$. Observe now that the pushdown query for Q implements the pushdown automata $PD(Q)$ in such a way that it recognizes only the strings α for which there exists a path spelling α on P and leaving b. Then we conclude that a constant c belongs to $A(Q)$ if and only if it belongs to $A(\hat{Q})$. □

We point out that a naive execution of the rewritten program can be inefficient or even non-terminating for cyclic databases. In Section 6 we shall present a technique, based on the approach of [14], where lists implementing pushdown stores, are represented as pairs consisting of the head and a pointer to the tuple storing the tail of the list. In this way, each possible cyclic sequence in the pushdown store is recorded only once and, therefore, termination is guaranteed.

3.1. Right-Linear Programs

As pointed out previously, the pushdown method is based on constructing a particular pushdown automaton to recognize a context-free language. Now, let us 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 actually acts as a finite state automaton. Indeed, if the program is right-linear, the pushdown store is either empty or contains only one symbol. Therefore, it is possible to delete the pushdown store and to put the information of the pushdown store into the state.

For a right-linear chain query, it is also possible to generate directly the pushdown query $\hat{Q}$ that emulates the finite state automaton. Thus, given a chain right-linear query $Q = \langle p(b, Y), P\rangle$ the pushdown query $\hat{Q}$ is equal to $\langle p_F(Y), \hat{P}\rangle$ where $\hat{P}$ consists of a fact of the form

$$q(b).$$

plus a rule of the form

$$q'(Y) \leftarrow q(X),\ a(X, Y)$$

for each production rule of the form $q \rightarrow a\,q'$ in $G(Q)$ with q and q' mutually recursive, and

$$q_F(Y) \leftarrow q(X),\ a(X, Y)$$

for each non-recursive production rule of the form $q \to a$ in $G(Q)$.

Example 2 Consider the following chain query $Q = \langle p(x_0, Y), P \rangle$, where P is:

$$\begin{array}{ll} \mathtt{p(X,Y)} \leftarrow & \mathtt{b(X,Y).} \\ \mathtt{p(X,Y)} \leftarrow & \mathtt{a(X,Z),\ \ p(Z,Y).} \end{array}$$

The grammar $G(Q)$ is regular right-linear and is as follows:

$$p \to b \mid a\ p$$

The pushdown automaton recognizing $L(Q)$ is as follows

	a	b	ϵ
(q_0, Z_0)			$(q, p\, Z_0)$
(q, p)	(q, p)	(q, ϵ)	

The pushdown query of Q is $\hat{Q} = \langle q(Y, [\,]), \hat{P} \rangle$ with $\hat{P}$ as follows:

$$\begin{array}{ll} \mathtt{q(x_0,[p]).} & \\ \mathtt{q(Y,[p])} \leftarrow & \mathtt{q(X,[p]),\ \ a(X,Y).} \\ \mathtt{q(Y,[\,])} \leftarrow & \mathtt{q(X,[p]),\ \ b(X,Y).} \end{array}$$

By deleting the pushdown store and putting its information into the state we obtain the following query $\hat{Q} = \langle q(Y), \hat{P} \rangle$ where $\hat{P}$ as follows:

$$\begin{array}{ll} \mathtt{q_p(x_0).} & \\ \mathtt{q_p(Y)} \leftarrow & \mathtt{q_p(X),\ \ a(X,Y).} \\ \mathtt{q(Y)} \leftarrow & \mathtt{q_p(X),\ \ b(X,Y).} \end{array}$$

Observe that the language $L(Q)$ can be recognized by the finite state automaton $FA(Q)$ whose transition function is as follows

$$\begin{array}{lll} \delta(p, a) & \to & p \\ \delta(p, b) & \to & p_F \end{array}$$

where p and p_F denote the initial and the final states, respectively. The new query is $\langle p_F(Y), \hat{P} \rangle$ with $\hat{P}$ as follows:

$$\begin{array}{ll} \mathtt{p(b).} & \\ \mathtt{p(Y)} \leftarrow & \mathtt{p(X),\ \ a(X,Y).} \\ \mathtt{p_F(Y)} \leftarrow & \mathtt{p(X),\ \ b(X,Y).} \end{array}$$

□

Corollary 1 *Let Q be a* **q***-chain query such that $G(Q)$ is right-linear. Then, the finite-state query of Q is equivalent to Q.*

Proof. This follows directly from Theorem 1 by inserting the pushdown store information into the state symbol and deleting the pushdown store. □

Thus, for right-linear queries the pushdown method does not use any pushdown store; i.e., given a right-linear query Q, the pushdown query of Q reduces to the finite state query of Q.

4. Grammar Transformations to improve Pushdown

In this section, we show that this kind of automaton becomes more effective when the grammar of the language has a particular structure. Furthermore, we show that programs where the grammar does not have this structure can be rewritten so that the corresponding grammar achieves the desired structure; also, this rewriting is based on known techniques for transforming grammars, particularly, those used to achieve the $LL(1)$ format [3].

Observe that if the grammar $G(Q)$ is regular left-linear then the pushdown method does not emulates a finite state automaton, as for the case where $G(Q)$ is regular right-linear, 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 where a predicate symbol $s \in \mathbf{q}$ is in the head of some left-recursive chain rule—let us call such an s *left-recursive*. Then, the definition $def(s)$ consists of $m > 0$ left-recursive chain rules and n chain rules that are not left-recursive (obviously we must have that $n > 0$ or s would not be satisfied):

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

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

$$\begin{array}{lcll} s & \rightarrow & \alpha_i & 1 \leq i \leq n \\ s & \rightarrow & s'\ \beta_i & 1 \leq i \leq m \end{array}$$

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 now apply the known transformations to remove left-recursion from the second group of rules for all left-recursive predicate symbols s, and then we rewrite the corresponding rules accordingly. It turns out that the resulting program, denoted by $can(P)$, does not contain any left-recursive **q**-chain — here $can(P)$ stands for *canonical format of P*.

Example 3 Left-Linear Transitive Closure. *Consider the following right-linear* **q***-chain query* $Q = \langle \mathtt{path(b,Y)}, P \rangle$*, where* $\mathbf{q} = \{path\}$ *and* P *is:*

$$\begin{array}{ll} \mathtt{path(X,Y)} \leftarrow & \mathtt{arc(X,Y)}. \\ \mathtt{path(X,Y)} \leftarrow & \mathtt{path(X,U)},\ \mathtt{arc(V,Y)}. \end{array}$$

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

$$\begin{array}{lcl} path & \rightarrow & arc\ path' \\ path' & \rightarrow & arc\ path' \mid \epsilon \end{array}$$

So, the program $can(P)$:

$$\begin{array}{ll} \mathtt{path(X,Y)} \leftarrow & \mathtt{arc(X,Z)},\ \mathtt{path'(Z,Y)}. \\ \mathtt{path'(X,X)} & \\ \mathtt{path'(X,Y)} \leftarrow & \mathtt{arc(X,Z)},\ \mathtt{path'(Z,Y)}. \end{array}$$

is right-linear; the pushdown query can be now solved efficiently. □

Example 4 Non-Linear Transitive Closure. *Assume now that the program P of the query of Example 3 is defined as:*

$$\begin{aligned}\mathtt{path(X,Y)} &\leftarrow \ \mathtt{arc(X,Y)}.\\ \mathtt{path(X,Y)} &\leftarrow \ \mathtt{path(X,U)},\ \mathtt{path(V,Y)}.\end{aligned}$$

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

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

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

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

Proposition 1 *Let $Q = \langle p(b,Y), P\rangle$ be a chain query and let $Q' = \langle p(b,Y), can(P)\rangle$. Then Q' is equivalent to Q.*

Proof. We have to show that a substitution Y/c is in $A(Q)$ if and only if it is also in $A(Q')$. The two languages $L(Q)$ and $L(Q')$ are equivalent, i.e., they consist of the same set of strings [3]. Therefore, also the set of paths starting from b and spelling a string $\alpha \in L(Q)$ on P and $can(P)$ coincide. □

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

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

Then, 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 the last recursive goal in the the rule. Obviously, if the resulting rule still has multiple recursive goals at its end, we repeat the transformation. The program obtained after performing this above transformations for all the predicate symbols s in P is denoted by $simple(P)$.

Proposition 2 *Given a chain query $Q = \langle p(b,Y), P\rangle$, Q is equivalent to $Q' = \langle p(b,Y),\ simple(P)\rangle$.*

Proof. $L(Q) = L(Q')$ and, therefore, the sets of paths starting from b and spelling a string $\alpha \in L(Q)$ on P and $simple(P)$ coincide. □

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:*

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

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:

$$\begin{array}{lll} s & \rightarrow & a\, s' \\ s' & \rightarrow & b\, s\, s' \mid \epsilon \end{array}$$

we can modify it into:

$$\begin{array}{lll} s & \rightarrow & a\, s' \\ s' & \rightarrow & b\, a\, s'\, s' \mid \epsilon \end{array}$$

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

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

$$\begin{array}{ll} \mathtt{path(X,Y)} \leftarrow & \mathtt{yellow(X,Y)}. \\ \mathtt{path(X,Y)} \leftarrow & \mathtt{path(X,U)},\ \ \mathtt{red(U,V)},\ \ \mathtt{path(V,W)},\ \ \mathtt{blue(W,Z)},\ \ \mathtt{path(Z,Y)}. \end{array}$$

We have that $can(P)$ is:

$$\begin{array}{ll} \mathtt{path(X,Y)} \leftarrow & \mathtt{yellow(X,Z)},\ \ \mathtt{path'(Z,Y)}. \\ \mathtt{path'(X,X)}. & \\ \mathtt{path'(X,Y)} \leftarrow & \mathtt{red(X,U)},\ \ \mathtt{path(U,W)},\ \ \mathtt{blue(W,Z)},\ \ \mathtt{path(Z,T)},\ \ \mathtt{path'(T,Y)}. \end{array}$$

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':

$$\begin{array}{ll} \mathtt{path(X,Y)} \leftarrow & \mathtt{yellow(X,Z)},\ \ \mathtt{path'(Z,Y)}. \\ \mathtt{path'(X,X)}. & \\ \mathtt{path'(X,Y)} \leftarrow & \mathtt{red(X,U)},\ \ \mathtt{yellow(U,V)},\ \ \mathtt{path'(V,W)},\ \ \mathtt{blue(W,Z)}, \\ & \mathtt{yellow(Z,T)},\ \ \mathtt{path'(T,S)},\ \ \mathtt{path'(S,Y)}. \end{array}$$

We can now apply the transformation *simple* to $\mathtt{path'}$ and the last rule of P' becomes:

$$\begin{array}{ll} \mathtt{path'(X,Y)} \leftarrow & \mathtt{red(X,U)},\ \ \mathtt{yellow(U,V)},\ \ \mathtt{path'(V,W)},\ \ \mathtt{blue(W,Z)}, \\ & \mathtt{yellow(Z,T)},\ \ \mathtt{path'(T,Y)} \end{array}$$ □

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

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

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

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

We rewrite the above rules as follows:

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

We now replace all 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 [13, 12], the pushdown method becomes much more efficient when applied to $simple'(P)$ rather than to P.

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

Proof. As for the proof of Proposition 2, $L(Q) = L(Q')$ and, therefore, the sets of paths starting from b and spelling a string $\alpha \in L(Q)$ on P and $simple'(P)$ coincide. □

Example 7 *Consider the* $\{path\}$*-chain query* $Q = \langle \texttt{path(b,Y)}, P\rangle$ *where P is defined as follows:*

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

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

```
path(X,Y) ←   path'(X,Z), yellow(Z,Y).
path'(X,X).
path'(X,Y) ←  red(X,V), path'(V,W), yellow(W,T), path'(T,Y).
```

As discussed in the next section, 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 **q**-chain query, the pushdown store can be efficiently implemented as follows whenever it contains strings of the form $\alpha^k(\beta)^n$, with $0 \leq k \leq 1$ and $n \geq 0$. 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. The next example illustrates that the above implementation of the pushdown store corresponds to applying the counting method.

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

$$\begin{array}{ll} \mathtt{sg(X,Y)} \leftarrow & \mathtt{c(X,Y).} \\ \mathtt{sg(X,Y)} \leftarrow & \mathtt{a(X,X1),\ \ sg(X1,Y1),\ \ b(Y1,Y).} \end{array}$$

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

$$\begin{array}{ll} \mathtt{q(d,[\,sg\,]).} & \\ \mathtt{q(Y,[\,sg,b\,|\,T\,])} \leftarrow & \mathtt{q(X,[\,sg\,|\,T\,]),\ \ a(X,Y).} \\ \mathtt{q(Y,T)} \leftarrow & \mathtt{q(X,[\,sg\,|\,T\,]),\ \ c(X,Y).} \\ \mathtt{q(Y,T)} \leftarrow & \mathtt{q(Y,[\,b\,|\,T\,]),\ \ b(X,Y).} \end{array}$$

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:

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

These rules are the same as those generated by the counting method. Obviously the query goal is $\mathtt{q_b(Y,0)}$. □

We now show that the above counting implementation of the pushdown store can be also used 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* $Q = \langle \mathtt{path(b,Y)}, P\rangle$ *where* P *is:*

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

Using the counting implementation of the pushdown store, we obtain the following program:

$$\begin{array}{ll} \mathtt{q_{path}(b,0)} & \\ \mathtt{q_{yellow}(X,I)} \leftarrow & \mathtt{q_{path}(X,I).} \\ \mathtt{q_{path}(Y,I+1)} \leftarrow & \mathtt{q_{path}(X,I),\ \ red(X,Y).} \\ \mathtt{q_{path}(Y,I-1)} \leftarrow & \mathtt{q_{yellow}(X,I),\ \ yellow(X,Y).} \end{array}$$

The query goal is $\mathtt{q_{yellow}(Y,0)}$. Observe that the above program cannot be handled by the counting method. □

Note that also the query of Example 6 can be optimized by using counters to implement the pushdown store since after elimination of right recursion it has a the same format of the query of Example 9.

6. Implementation and Termination

As pointed out in Section 3, the pushdown method could be inefficient or even non-terminating for cyclic databases. In this section we present how the method is implemented in order to guarantee efficiency and termination.

The basic idea is to 'distribute' stores among tuples and to link the tuples which are used to memorize the same store. More specifically, the store associated with a tuple is memorized by means of two distinct elements: a list containing a *block* of elements in the top of the store and a *link* to a tuple which can be used to derive the tail of the store. Thus, a tuple of the form $q(x, [p_1, ..., p_n])$ is memorized as $q(x, [p_1, ..., p_k], Id)$, where $k \leq n$, and Id is a link to some tuple which permits to determine the tail $[p_{k+1}, ..., p_n]$ of the store. Let us now present how the pushdown method is implemented.

Let $Q = \langle p(a, Y), P \rangle$ be a query and let $\hat{Q} = \langle q(Y, [\,]), P' \rangle$ be the pushdown query of Q. The pushdown implementation query of Q, denoted $I(\hat{Q})$, is the pushdown query $\langle q(Y, [\,], _), P" \rangle$ where $P"$ is derived from P' as follows:

1. A fact of the form

$$q(b, [p]).$$

   is substituted by the following fact where nil is a new constant

$$q(b, [p], nil).$$

2. A rule r^j of the form

$$q(Y, [p^j_1, a^j_1, ..., p^j_n, a^j_n|T]) \leftarrow q(X, [p^j_0|T]), \quad a^j_0(X, Y)$$

   is substituted by the rule

$$q(Y, [p^j_1, a^j_1, ..., p^j_n, a^j_n], Id(X)) \leftarrow q(X, [p^j_0|T], I), \ \mathbf{a}^j_0(X, Y).$$

   where $Id(X)$ is a unique identifier associated with the list of ground tuples having X as first argument. In the following, for the sake of simplicity we assume that $Id(X) = X$.

3. A rule r^j of the following form with $i < n$ (n is the number of base conjunction in the rule)

$$q(Y, T) \leftarrow q(X, [a^j_i|T]), \ \mathbf{a}^j_i(X, Y)$$

   is substituted by the rule

$$q(Y, T, I) \leftarrow q(X, [a^j_i|T], I), \ \mathbf{a}^j_i(X, Y).$$

4. A rule r^j of the form

$$q(Y,T) \leftarrow q(X,[a_n^j|T]),\ \mathbf{a}_n^j(X,Y).$$

where a_n^j denotes the last base conjunction in the rule r^j, is substituted by the rule

$$q(Y,T,I) \leftarrow q(X,[a_n^j],Id(Z)),\ q(Z,[p|T],I),\ \mathbf{a}_n^j(X,Y).$$

The following example should clarify how the pushdown implementation query is built.

Example 10 Consider the query $Q = \langle \mathtt{sg(b,Y)}, P\rangle$ of Example 1. The pushdown program $\hat{Q}$, presented in Example 1 of Section 3, is as follows:

$$\begin{array}{ll}
\mathtt{q(b,[\,sg\,]).} & \\
\mathtt{q(Y,T)} \leftarrow & \mathtt{q(X,[\,sg\,|\,T\,]),\ \ a(X,Y).} \\
\mathtt{q(Y,[\,sg,c,sg,d\,|\,T\,])} \leftarrow & \mathtt{q(X,[\,sg\,|\,T\,]),\ \ b(X,Y).} \\
\mathtt{q(Y,T)} \leftarrow & \mathtt{q(X,[\,c\,|\,T\,]),\ \ c(X,Y).} \\
\mathtt{q(Y,T)} \leftarrow & \mathtt{q(X,[\,d\,|\,T\,]),\ \ d(X,Y).}
\end{array}$$

The pushdown program $I(\hat{Q})$ is $\langle sg(Y,[\,],nil), P\text{"}\rangle$ where the program P" is as follows:

$$\begin{array}{ll}
\mathtt{q(b,[\,sg\,],nil).} & \\
\mathtt{q(Y,T,I)} \leftarrow & \mathtt{q(X,[\,sg\,|\,T\,],I),\ \ a(X,Y).} \\
\mathtt{q(Y,[\,sg,c,sg,d\,],X)} \leftarrow & \mathtt{q(X,[\,sg\,|\,_\,],_),\ \ b(X,Y).} \\
\mathtt{q(Y,T,I)} \leftarrow & \mathtt{q(X,[\,c\,|\,T\,],I),\ \ c(X,Y).} \\
\mathtt{q(Y,T,I)} \leftarrow & \mathtt{q(X,[\,d\,],Z),\ \ q(Z,[\,sg\,|\,T\,],I),\ \ d(X,Y).}
\end{array}$$

□

Proposition 4 *Let Q be a chain query and let $\hat{Q}$ be the pushdown query of Q. Then $I(\hat{Q})$ is equivalent to Q.*

Proof. The query $I(\hat{Q})$ is just a different implementation of the pushdown automata $PD(Q)$ and, therefore, it recognizes only the strings α for which there exists a path spelling α on P and leaving b. Then we conclude that a constant c belongs to $A(Q)$ if and only if it belongs to $A(I(\hat{Q}))$. □

Moreover, the implementation technique, besides efficiency, guarantees also termination of the evaluation process.

Proposition 5 *Let Q be a chain query and let $\hat{Q}$ be the pushdown query of Q. The bottom-up computation of $I(\hat{Q})$ always terminates.*

Proof. The number of constants as well as the number of blocks is finite and, therefore, the number of links to tuples is also finite. Therefore, the number of tuples is finite and this implies that the fixpoint computation always terminates. □

The following example shows how queries are computed in the presence of cyclic databases.

Example 11 Let $I(\hat{Q}) = \langle sg(Y, [\,], nil), P" \rangle$ be the pushdown query of the previous Example where the binding b has been substituted by the constant 1. Consider the database pictured in the following Figure 3 where a tuple (x, y) of a relation r is represented by an arc from x to y with label r:

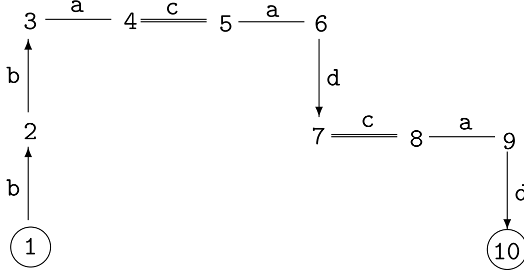

Figure 3 — Acyclic database

The evaluation of the program P" produces the following tuples q(1, [sg], nil), q(2, [sg, c, sg, d], 1), q(3, [sg, c, sg, d], 2), q(4, [c, sg, d], 2), q(5, [sg, d], 2), q(6, [d], 2), q(7[c, sg, d], 1), q(8, [sg, d], 1), q(9, [d], 1) and q(10, [ ], nil). Therefore, the answer is $Y = 10$.

Consider now cyclic database pictured in Figure 4.

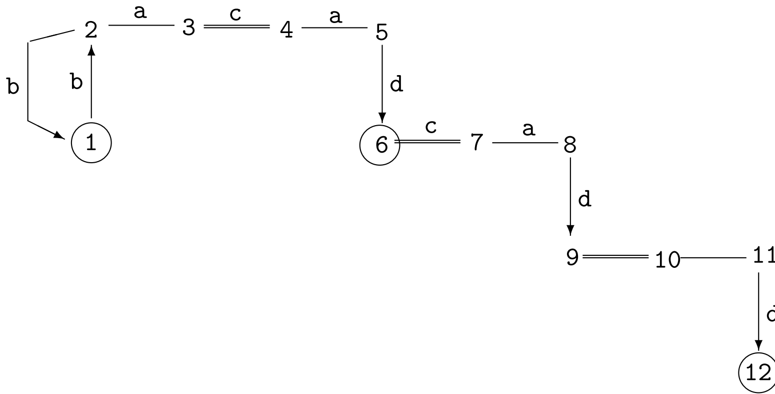

Figure 4 — Cyclic database

The evaluation of the program P" produces the tuples q(1, [sg], nil), q(2, [sg, c, sg, d], 1), q(1, [sg, c, sg, d], 2), q(3, [c, sg, d], 1), q(4, [sg, d], 1), q(5, [d], 1), q(6, [ ], nil), q(6, [c, sg, d], 2), q(7, [sg, d], 2), q(8, [d], 2), q(9, [c, sg, d], 1), q(10, [sg, d], 1), q(11, [d], 1) and q(10, [ ], nil). Therefore, the answers are $Y = 6$ and $Y = 12$. □

We conclude this section by pointing out that when it is not possible to apply one of the reductions presented in Sections 4 and 5, then we implement our method by means of pointers and shared structures as described above. The implementation of the pushdown method can be seen as a smart implementation of the supplementary magic-set method [27] (see also [8]). Moreover, there is an important difference

for non-linear programs since our method generates fewer non-linear recursive rules than the supplementary magic-set method. Take, for instance, the non-linear query of Example 1. The program obtained by applying the supplementary magic-set method is as follows:

$$
\begin{array}{ll}
\texttt{m_sg(b)}. & \\
\texttt{m_sg}(\texttt{X}_1) \leftarrow & \texttt{s_sg}_1(_, \texttt{X}_1). \\
\texttt{m_sg}(\texttt{Y}_2) \leftarrow & \texttt{s_sg}_3(_, \texttt{Y}_2). \\
\texttt{s_sg}_1(\texttt{X}, \texttt{X}_1) \leftarrow & \texttt{m_sg}(\texttt{X}),\ \texttt{b}(\texttt{X}, \texttt{X}_1). \\
\texttt{s_sg}_2(\texttt{X}, \texttt{X}_2) \leftarrow & \texttt{s_sg}_1(\texttt{X}, \texttt{X}_1),\ \texttt{sg}(\texttt{X}_1, \texttt{X}_2). \\
\texttt{s_sg}_3(\texttt{X}, \texttt{X}_3) \leftarrow & \texttt{s_sg}_2(\texttt{X}, \texttt{X}_2),\ \texttt{c}(\texttt{X}_2, \texttt{Y}_2). \\
\texttt{sg}(\texttt{X}, \texttt{Y}) \leftarrow & \texttt{m_sg}(\texttt{X}),\ \texttt{a}(\texttt{X}, \texttt{Y}). \\
\texttt{sg}(\texttt{X}, \texttt{Y}) \leftarrow & \texttt{s_sg}_3(\texttt{X}, \texttt{Y}_2),\ \texttt{sg}(\texttt{Y}_2, \texttt{Y}_1),\ \texttt{d}(\texttt{Y}_1, \texttt{Y}).
\end{array}
$$

All rules but the first one are mutually recursive but the rule defining the predicate $\texttt{s_sg}_2$ and the second rule defining the predicate sg are by-linear, i.e., they have two occurrences of predicates mutually recursive with the head predicate. The program generated by the pushdown method contains only one bi-linear rule and, therefore, its execution can be more efficient.

In the general case of a non-linear recursive rule r having $n > 1$ recursive predicates in its body, the pushdown method generates only one bi-linear rule, whereas the supplementary magic-set method generates n bi-linear rules. Moreover, the space used by the pushdown method is less than that used by the supplementary magic-set method since our method does not use *magic* predicates.

7. Experimental Results

To better understand the differences between the pushdown method and others classical methods such as the magic-set and the supplementary magic-set methods, we have performed some experiments using two different queries.

The first query is $Q_1 = \langle \texttt{p(b,Y)}, P_1 \rangle$ where P_1 is the following bi-linear program:

$$
\begin{array}{ll}
\texttt{p(X,X)}. & \\
\texttt{p(X,Y)} \leftarrow & \texttt{a(X,U)},\ \texttt{p(U,V)},\ \texttt{b(V,W)},\ \texttt{p(W,Y)}.
\end{array}
$$

Observe that this program is the same as the one reported in Example 9 and, therefore, we can use counters to implement pushdown stores[a].

The second query we consider is $Q_2 = \langle \texttt{p(b,Y)}, P_2 \rangle$ where P_2 is the following non-linear program:

$$
\begin{array}{ll}
\texttt{p(X,Y)} \leftarrow & \texttt{c(X,Y)}. \\
\texttt{p(X,X)} \leftarrow & \texttt{a(X,X1)},\ \texttt{p(X1,X2)},\ \texttt{a(X2,X3)},\ \texttt{p(X3,Y3)},\ \texttt{b(Y3,Y2)},\ \texttt{p(Y2,Y1)},\ \texttt{b(Y1,Y)}.
\end{array}
$$

In this case, pushdown stores are implemented by means of pointers.

We have assumed that the base relations A, B and C, associated, respectively, with the predicates a, b and c, have a "cylindric structure" [6]. We represent the

[a] We are assuming that it is known that the database is acyclic. A database associated with a chain query can be represented by means of a digraph and checking if a graph is acyclic can be done very efficiently [4]

database by means of a digraph G, called query-graph, which is partitioned into the three sub-graphs G_A, G_B and G_C (see Figure 4). For each tuple (x, y) in the relation A (resp., B, C) there is an arc going up (resp. going down, flat) in G. The sets of arcs in A (resp. B, C) identify the sub-graph G_A (resp. G_B, G_C). The nodes are arranged into layers and the graph G has h layers (h denotes the height of the graph). All layers in G_A and G_B have the same number b (base) of nodes. Thus, each node G is identified by a pair (i, j) where j $(0 \leq j \leq h-1)$ denotes the layer and i $(0 \leq i \leq b-1)$ denotes the position in the layer.

Each node (i, j) in G_A (resp. G_B) with $0 \leq j < h-1$ (resp. $0 < j \leq h-1$) is the source of g arcs, whereas each node (i, j) in G_A (resp. G_B) with $0 < j \leq h-1$ (resp. $0 \leq j < h-1$) is the sink of g arcs. In particular, for each node (i, j) in G_A such that $j < h-1$ and for each integer k between 0 and $g-1$, there is an arc in G_A from the node (i, j) to the node $((i + k \times (b \ div \ g)) \ mod \ b, j+1)$. For instance, assuming $h = 4$, $b = 4$ and $g = 2$, from the node $(2, 2)$ there are two arcs to the nodes $(2, 3)$ and $(0, 3)$, respectively. Analogously, for each node (i, j) in G_B such that $(j > 0)$ and for each integer k between 0 and $g-1$ there is an arc in G_B from the node (i, j) to the node $((i + k(b \ div \ g) \ mod \ b), j-1)$.

The sub-graph G_C (used only in the query Q_2) contains $(h \times c)$ arcs, where c $(\leq b)$ denotes the number of arcs connecting nodes in the level j $(0 \leq j \leq h-1)$ of G_A with nodes in the same level of G_B. The c arcs connecting nodes in G_A with nodes in G_B are uniformly distributed.

The binding b in the query goal denotes a node of a fixed layer in the subgraph G_A. The query graph with parameters $b = 4$, $h = 4$, $g = 2$ and $c = 2$, where only the arcs of G_C connecting nodes in the layer 3 are reported, is pictured in Figure 4.

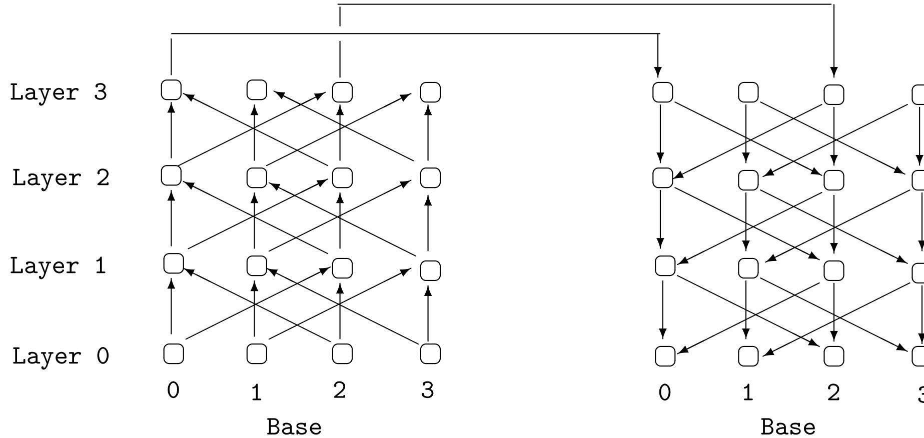

Figure 4: Query graph associated with a cylindric database

For the experiments reported below, we have considered a database D whose query-graph has a "cylindric structure" with $b = 15$, $h = 20$, $g = 3$ and $c = 4$. Thus, the query-graph G contains 300 nodes and 1350 arcs. More specifically, the sub-graphs G_A and G_B have both 300 nodes and 635 arcs whereas the sub-graph

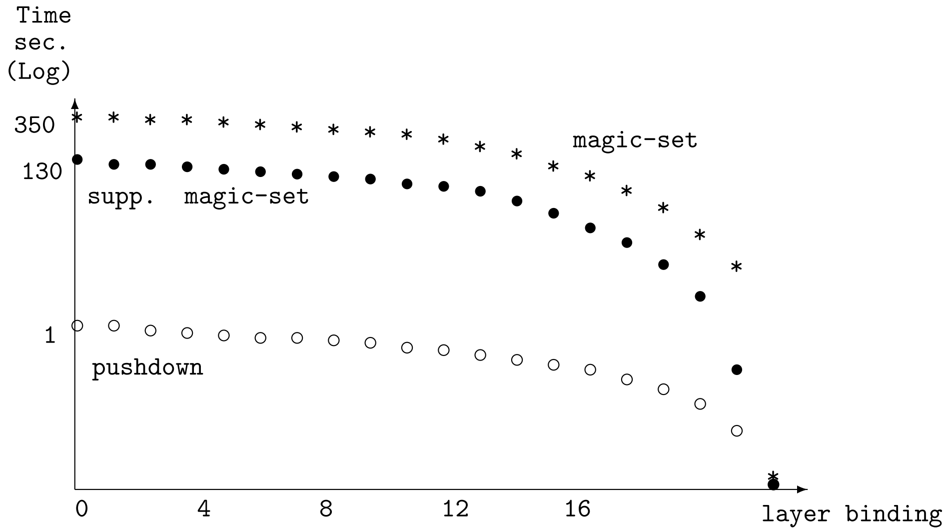

Figure 5: Experimental results for the query Q_1 with $b = 15$, $h = 20$, $g = 3$, $c = 4$

G_C (used only in the query Q_2) has 60 arcs. In our experiments we have considered different bindings (one for each layer) and a fixed database.

The results for the queries Q_1 and Q_2 are reported in Figures 5 and 6, respectively.

The evaluation of queries has been carried out by means of programs emulating the bottom-up computation of the (rewritten) queries. Such programs have been implemented in Pascal and run on personal computer. The results of the experiments have been performed by using a PC with CPU Intel Pentium, 16 Mbyte of Ram and clock of 166 MHz.

Concluding, we observe that

1. For the query Q_1, the counter-based implementation is very efficient and there are, on the average, two orders of magnitude between the supplementary magic-set and the pushdown methods.

2. The pointer-based implementation has the same performances of the supplementary magic-set if we consider bindings in high layers. (As expected since a small number of arcs in the query-graph is used for the evaluation of the queries.) But, when we consider bindings in low layers of the query-graph, the pushdown method is, on the average, between four and five times faster. Thus, for large databases, the pushdown method performs significantly better than the supplementary magic-set method.

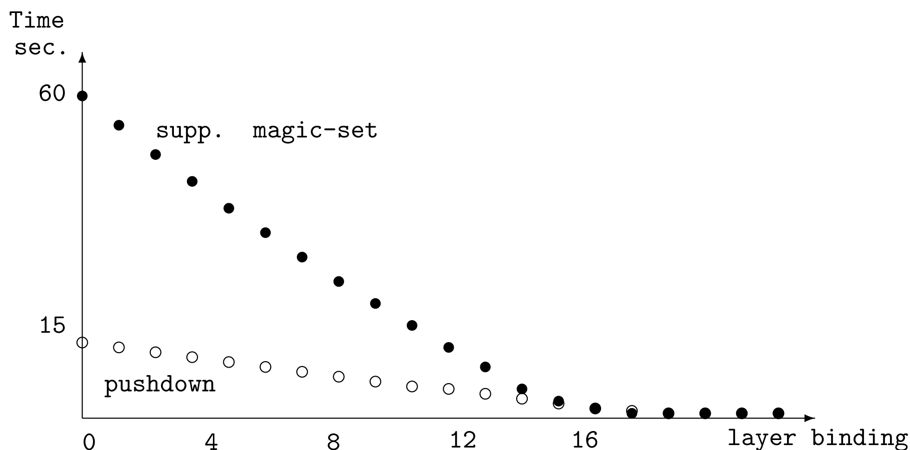

Figure 6: Experimental results for the query Q_2 with $b = 15$, $h = 20$, $g = 3$, $c = 4$

8. Conclusions

In this paper, we have presented a method, called *pushdown method*, which transforms chain programs into programs that emulate pushdown automata. The method build on the analogy between chain programs and context-free languages that are recognized by a particular scheme of pushdown automata In the proposed approach, the automaton is implemented as a factorized left-linear logic program which uses lists to emulates pushdown stores. The pushdown 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 that are query-equivalent to linear ones.

The method here presented has been implemented and its performance compared to those of other techniques. In our implementation, a list-term emulating the pushdown store is split into two terms, a list-term and a constant term denoting the tuple storing the tail of the list. Then, the resulting program contains one bi-linear rule (i.e., a rule with exactly two mutually recursive predicates) for each non-linear rule in the original program. This modified re-writing controls the data complexity and ensures terminating computations for the case of cyclic data. Preliminary experimental results obtained with this implementation are encouraging inasmuch as it appears that the pushdown method has better performance than classical methods, such as the magic set method. For bound queries, the pushdown method also behaves better than Yannakakis' method, which has a better worst-case complexity, but cannot use bindings to narrow the search space.

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*, 1987, pages 210–213.
3. A.V. Aho, and J.D. Ullmann. *The Theory of Parsing Translating and Compiling.* Volume 1 & 2, Prentice-Hall, 1972.
4. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and analysis of Computer Algorithms*, Addison-Wesley, 1974.
5. F. Bancilhon, D. Mayer, Y. Sagiv, and J.F. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the Fifth ACM PODS*, 1986, pages 1–15.
6. F. Bancilhon and R. Ramakrisnhan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufman, Los altos, CA, 1988, pages 439–518.
7. C. Beeri, P. Kanellakis, F. Bancilhon, and R. Ramakrisnhan. Bounds on the Propagation of Selection into Logic Programs, *J. of Computer and System Science*, Vol. 41, No. 2, Oct. 1990, pages 157–180.
8. C. Beeri and R. Ramakrisnhan. On the power of magic. *Journal of Logic Programming*, 10 (3 & 4), 1991, pages 255–299.
9. G. Dong, On Datalog Linearization of Chain Queries. In J.D. Ullman, editor, *Theoretical Studies in Computer Science*, Academic Press, 1991, pages 181–206.
10. G. Dong, Datalog Expressiveness of Chain Queries: Grammar Tools and Characterization. In *Proc. of the Eleventh ACM Symp. on Principles of Database Systems*, 1992, pages 81–90.
11. J. Han. Selection of Processing Strate gies for Different Recursive Queries. In *Proceedings Internatio nal Conference on Data and Knowledge Bases*, Jerusalem, Israel, 1988.
12. S. Greco, D Saccà and C. Zaniolo, The Pushdown Method to Optimize Chain Logic Queries. In *Proc. International Colloquium on Automata Languages and Programming*, 1995.
13. S. Greco, and E. Spadafora. Implementation of chain queries. *Proc. APPIA-GULP-PRODE Int. Conf. on Declarative Programming.* San Sebastian, 1996.
14. S. Greco and C. Zaniolo, Optimization of linear logic programs using counting methods. In *Proceedings of the Extending Database Technology*, 1992, pages 187–220.
15. R. Haddad and J. Naughton, A counting algorithm for a cyclic binary query. *Journal of Computer and System Science*, 43(1):145-169, 1991.
16. Y.E. Ioannidis and E. Wong. Transforming non linear recursion to linear recursion. In *Proceedings Second Int. Conf. on Expert Database systems* (L. Kerschberg ed.), 1988, pages 187–207.
17. Y.E. Ioannidis. Commutativity and its Role in the Processing of Linear Recursion. *Journal of Logic programming* Vol. 14, No. 3-4, 1992, pages 223–252.
18. B. Lang. Datalog Automata. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases* Jerusalem, Israel, March, 1988, pages 389–401.
19. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 2nd edition, 1987.
20. 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.
21. S. J. Filkenstein, N. Mattos, I. S. Mumick and H. Pirahesh. Expressing Recursive Queries in SQL. *ISO-IEC JTC1/SC21 WG3 DBL MCi Technical Report*, march, 1996.
22. J. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.F. Ullman. Argument Reduction by Factoring. In *Proc. of the 15th Conf. on Very Large data Bases*, 1989, pages

173–182.
23. J. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.F. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the 1988 ACM SIGMOD Conference*, 1989, pages 235–242.
24. J.E. Hopcroft, and J.F. Ullmann. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
25. F.C.N. Pereira, and D.H.D. Warren. Definite Clause Grammars for Language Analysis - A Survey on the Formalism and a Comparison with Augmented Transition networks. *Artificial Intelligence*, No. 13, 1980, pages 231–278.
26. R. Ramakrisnhan, Y. Sagiv, J.F. Ullman, and M.Y. Vardi. Logical Query Optimization by Proof-Tree Transformation. In *Journal of Computer and System Science*, No. 47, pages 222-248, 1993.
27. 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.
28. D. Saccà and C. Zaniolo, Magic-counting methods. In *Proceedings of the 1987 ACM SIGMOD Int. Conf. on Management of Data*, pages 149-59, 1987.
29. 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.
30. Y.P. Saraiya. Linearizing nonlinear recursions in polynomial time In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, 1989, pages 182–189.
31. J.F. Ullmann. *Principles of Data and Knowledge-Base Systems.* Volume 1 & 2, Computer Science Press, New York, 1989.
32. J.F. Ullmann. The Interface Between Language Theory and Database Theory. In J.D. Ullman, editor, *Theoretical Studies in Computer Science*, Academic Press, 1991, pages 133–151.
33. J.F. Ullmann and A. Van Gelder. Parallel Complexity of Logical Query Programs. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 1986, pages 438–454.
34. L. Vielle. Recursive Query processing: The Power of Logic. *Theoretical Computer Science.* No. 69, 1989, pages 1–53.
35. P.T. Wood. Factoring Augmented Regular Chain Programs. *Proc. of the 16th VLDB Conference.* Brisbane, Australia, 1990, pages 225–263.
36. M. Yannakakis. Graph-Theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990, pages 230–242.
37. W. Zang, C.T. Yu and D. Troy. Linearization of Nonlinear Recursive Rules. *IEEE Transaction on Software Engineering*, Vol. 15, No. 9, September 1989, pages 1109–1119.