Implementation of Chain Logic Queries *

Sergio Greco, Eugenio Spadafora
Dip. Elettronica Informatica e Sistemistica
Università della Calabria
87030 Rende, Italy
<lastname>si.deis.unical.it

Abstract

The problem of finding efficient implementations for recursive queries with bound arguments has been deeply investigated in the last years. Different 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 have been proposed. A method, called *pushdown method*, based on the analogy of chain program, context-free languages and pushdown automata, has been recently proposed for the class of chain programs. However, this method can be inefficient or even non terminating. In this paper we propose an implementation technique for the pushdown method which guaranties efficiency and terminating computations. We present some experimental results which show that queries can be computed more efficiently by using the pushdown method with respect to other methods such as the magic set and the supplementary magic set methods.

1 Introduction

The problem of finding efficient implementations for recursive queries with bound arguments has been deeply investigated in the last years. Different 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 have been proposed [6, 10, 14, 15, 18, 19]. 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 rules, in a chain-like fashion [5, 7, 23]. For these queries, insisting on general optimization methods (e.g., the *magic-set* method [19]) 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

*Work partially supported by a European Union grant under the EU-US project "DEUS EX MACHINA: non-determinism for deductive databases" and by a MURST project "Sistemi formali e strumenti per basi di dati evolute".

have been proposed in the literature (e.g., in [1, 5, 7, 23, 24]). 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, preserves the original simplicity and efficiency [19] only for a subset of chain queries whose recursive rules are linear.

A method, called *pushdown method*, based on the analogy of chain program, context-free languages and pushdown automata, has been recently proposed for the class of chain programs [9]. The method 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. The 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. Moreover, it generalizes and unifies previous techniques such as the 'counting' and 'right-, left-, mixed-linear' methods and it also succeeds in reducing many non-linear programs to query-equivalent linear ones.

However, a direct execution of the rewritten program can be inefficient or even non-terminating for cyclic databases. In this paper we shall present a technique, based on the approach of [10], 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. Further, we show that the implementation of the pushdown method gives better performances with respect to other classical methods such as the magic-set and the supplementary magic-set methods.

We recall that analogies between chain queries and context-free languages were investigated by several authors, including [5, 2, 7, 8, 17, 20, 21]. In particular, the use of automata to compute general logic queries was first proposed by Lang [12]. 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 [22], 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 [24], 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.

For space limitation the proofs of our theorems are omitted and they can be found in [11].

2 Preliminaries

We shall assume that the reader is familiar with basic definitions and concepts of logic programming [13] and of the DATALOG language [19]. 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 $A(Q) = A(Q')$.

We assume that the program has been partitioned according to a topological order $(P_1, ..., P_n)$ such that each two predicates p and q defined in the same component P_i are mutually recursive. This means that each predicate appearing in P_i depends only on predicates belonging to P_j such that $j \leq i$. We assume also that the computation follows the topological order and that when we compute the component P_i the components $P_1, ..., P_{i-1}$ have been already computed. When we compute the component P_i all the facts obtained from the computation of the components $P_1, ..., P_{i-1}$ are basically treated the same as database facts. A rule in a component P_i is called *exit rule* if each predicate in the body belongs to a component P_j such that $j < i$. All the other rules are *recursive rules*. For the sake of simplicity, from now on we assume that out programs consist of only one component.

Given a DATALOG (i.e., function-symbol free) program P, a rule of P is a *chain rule* if it has the following general format:

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

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 are not mutually recursive with p_0, and each p_i, $1 \leq i \leq n$, is a (not necessarily distinct) predicate symbol mutually recursive with p_0. 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 $n = 0$ (r is of the form $p_0(X_0, Y_0) \leftarrow \mathbf{a}_0(X_0, Y_0)$) then r is called an *exit chain rule*; moreover if $\mathbf{a}_0(X_0, Y_0)$ is the empty conjunction (r is of the form $p_0(X_0, X_0)$) then r is called an *elementary chain* rule. Otherwise (i.e., $n > 0$), it is called a *recurrence chain* rule. 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 *chain program* if for each recursive predicate symbol p, every rule in $def(p)$ is chain and for each two atoms $p(X, Y)$, $p(Z, W)$ occurring in the body or the head of chain rules, $X = Z$ and $Y = W$ modulo renaming of the variables, thus the binding is passed through any atom of the same recursive predicate symbol always using the same pattern.

A *bound chain query* Q, is a query $\langle p(b, Y), P\rangle$, where P is a chain program, p is a recursive predicate symbol, b is a list bound arguments and Y is a list of variables.

3 The Pushdown Method

The *pushdown method* is based on the analogy of chain queries and context-free grammars [20]. Without loss of generality we assume that lists of variables in the chain rules consist of one variable and that the first argument of the query goal is a constant whereas the second one is a variable. Thus, all predicates are binary.

Consider a chain query of the form $Q = \langle p(b, Y), P\rangle$. Let V be the set of all predicate symbols occurring in the chain rules, the set of recursive predicate symbols is the set V_N of non-terminal symbols and $V_T = V - V_N$. The query Q has associated a context-free language $L(Q)$ on the alphabet V_T defined by the grammar $G(Q) = \langle V_N, V_T, \Pi, p\rangle$ where the production rules in Π are as follows.

For each chain rule r_j of the form:

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

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 [16] 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 1 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 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).

In order to compute $A(Q)$, it is sufficient to use the automaton of Figure 1 to recognize all paths leaving from b and spelling a string α of $L(Q)$ on P [1]. This can

	$\mathbf{a}_0^j$	$\mathbf{a}_1^j$	$\cdots$	$\mathbf{a}_n^j$	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 1: *Pushdown Automaton recognizing $L(Q)$*

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 1. 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 is called the *pushdown method.*

Theorem 1: *[9] Let Q be a chain query. Then the pushdown-query of Q is equivalent to Q.* □

Example 1: Consider the query $Q = \langle \mathtt{sg(b, Y)}, P\rangle$ where P is as follows:

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

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

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

Observe that the rewritten program is not any-more DATALOG. □

We point out that a naive execution of the rewritten program can be inefficient or even non-terminating for cyclic databases.

Right-Linear Programs

As pointed out the pushdown method is based on constructing a particular pushdown automaton to recognize a context-free language.

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 can be either empty or it 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.

Of coarse, for right-linear chain queries, it is possible to generate directly the pushdown query $\hat{Q}$ which works as a finite state automata. Thus, given a chain right-linear query $Q = \langle p(x_0, Y), P\rangle$ the pushdown query $\hat{Q}$ is equal to $\langle q_F(Y), \hat{P}\rangle$ where $\hat{P}$ consists of the following rules

$q(x_0)$.
$q'(Y) \leftarrow \quad q(X),\ a(X,Y)$ $\qquad \forall$ *rec. rule* $q(X,Z) \leftarrow a(X,Y),\ q(Y,Z)$ *in* P
$q_F(Y) \leftarrow \quad q(X),\ a(X,Y)$ $\qquad \forall$ *exit rule* $q(X,Y) \leftarrow a(X,Y)$ *in* P

Thus, for right-linear queries the pushdown method does not use any pushdown store.

4 Implementation

In this section we present two techniques for the implementation of the pushdown method. The first technique uses counters to implements pushdown stores whereas the second one uses pointers. As showed next, the counter implementation is very efficient but it has a limited applicability. Counterwisely, the pointer based implementation is general but less efficient.

4.1 Counting implementation

We now present 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 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 2: Consider the linear program defining the same-generation with the query-goal sg(d, Y):

```
sg(X, Y) ← c(X, Y).
sg(X, Y) ← a(X, X1), sg(X1, Y1), b(Y1, Y).
```

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

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

```
qsg(d, 0).
qsg(Y, I) ← qsg(X, J), a(X, Y), I = J + 1.
qb(Y, I) ← qsg(X, I), c(X, Y).
qb(Y, I) ← qb(Y, J), b(X, Y), I = J − 1 J > 0.
```

These rules are the same as those generated by the counting method. Obviously the query goal is qb(Y, 0). □

The 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:

$$p(X_0, Y_2) \leftarrow a_0(X_0, Y_0),\ p(Y_0, X_1),\ a_1(X_1, Y_1),\ p(Y_1, Y_2)$$

Example 3: Red/yellow path. Consider the query $Q = \langle$path(b, Y), $P\rangle$ where P is:

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

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

```
qpath(b, 0)
qyellow(X, I) ← qpath(X, I).
qpath(Y, I + 1) ← qpath(X, I), red(X, Y).
qpath(Y, I − 1) ← qyellow(X, I), yellow(X, Y), I > 0.
```

The query goal is $q_{\texttt{yellow}}(\texttt{Y},\texttt{0})$. Observe that the above program cannot be handled by the counting method [19]. □

We point out that the counting implementation is not safe if the database is cyclic since the computation could be non terminating.

4.2 Pointer based Implementation

The basic idea is to implement pushdown stores by means of pointers. In particular, a pushdown store is substituted by a pair $(head, tail)$ where $head$ contains a block of symbols appearing on the top of the store and $tail$ is the identifier of a tuple containing the tail. In order to avoid non-terminating computations, the tuples having the same value for the first argument are grouped and have associated a unique identifier which can be implemented by means of a pointer. The identifier of a set of "grouped tuples" having x as first argument will be denoted $Id(x)$. The sequence of all symbols in the pushdown store can be determined by navigating the linked tuples.

Let $Q = \langle p(a, Y), P\rangle$ be a query and let $\hat{Q} = \langle q(Y, [\,]), \hat{P}\rangle$ be the pushdown query of Q. Then, we denote with $I(\hat{Q})$ the pushdown query $\langle q(Y, [\,], nil), I(\hat{P})\rangle$ where $I(\hat{P})$ is derived from $\hat{P}$ as follows:

1. A fact of the form: $q(b, [p])$ is substituted by the fact $q(b, [p], nil)$ where nil is a new constant.
2. A rule of the form

   $$q(Y, [p_1^j, a_1^j, ..., p_n^j, a_n^j|T]) \leftarrow q(X, [p_0^j|T]),\ \mathbf{a}_0^j(X, Y)$$

   is substituted by the rule

   $$q(Y, [p_1^j, a_1^j, ..., p_n^j, a_n^j], Id(X)) \leftarrow q(X, [p_0^j|W], I),\ \mathbf{a}_0^j(X, Y).$$

   where $Id(X)$ is the identifier of the set of tuples having X as first argument. Obviously, if X is a simple constant and we have only one recursive predicate then we can assume that $Id(X) = X$.
3. A rule of the form

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

   $i < n$, is substituted by the rule

   $$q(Y, W, I) \leftarrow q(X, [a_i^j|W], I),\ \mathbf{a}_i^j(X, Y)$$

4. A rule 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

   $$q(Y, W, I) \leftarrow q(X, [a_n^j], Id(Z)),\ q(Z, [p_0^j|W], I),\ \mathbf{a}_n^j(X, Y).$$

Theorem 2: *Let Q be a chain query and let $\hat{Q}$ be the pushdown query of Q. Then $I(\hat{Q})$ is equivalent to Q.* □

Theorem 3: *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.* □

The implementation of the pushdown method can be seen as a smart implementation of the supplementary magic set method [19]. Moreover, for non-linear programs there is an important difference, since the pushdown method generates a lesser number of non-linear rules w.r.t. the supplementary magic-set method. Take, for instance, the non-linear query of Example 1. The program generated by the supplementary magic-set method is as follows:

$$\begin{array}{ll}
\texttt{m_sg}(\texttt{1}). & \\
\texttt{m_sg}(\texttt{X}_1) \leftarrow & \texttt{s_sg}_1(_, \texttt{X}_1). \\
\texttt{m_sg}(\texttt{Y}_2) \leftarrow & \texttt{s_sg}_2(_, \texttt{Y}_2). \\
\texttt{s_sg}_1(\texttt{X}, \texttt{X}_1) \leftarrow & \texttt{m_sg}(\texttt{X}),\ \texttt{b}(\texttt{X}, \texttt{X}_1). \\
\texttt{s_sg}_2(\texttt{X}, \texttt{Y}_2) \leftarrow & \texttt{s_sg}_1(\texttt{X}, \texttt{X}_1),\ \texttt{sg}(\texttt{X}_1, \texttt{X}_2),\ \texttt{c}(\texttt{X}_2, \texttt{Y}_2). \\
\texttt{sg}(\texttt{X}, \texttt{Y}) \leftarrow & \texttt{m_sg}(\texttt{X}),\ \texttt{a}(\texttt{X}, \texttt{Y}). \\
\texttt{sg}(\texttt{X}, \texttt{Y}) \leftarrow & \texttt{s_sg}_2(\texttt{X}, \texttt{Y}_2),\ \texttt{sg}(\texttt{Y}_2, \texttt{Y}_1),\ \texttt{d}(\texttt{Y}_1, \texttt{Y}).
\end{array}$$

Observe that the rewritten program contains two non-linear rules whereas the program generated by the pointer-based pushdown method contains only one non-recursive rule. In the general case, given a non-linear recursive rule having n recursive predicates in its body, then the pushdown method generates only one bi-linear rule whereas the supplementary magic set method generates n bi-linear.

5 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, we have done some experiments using two different queries.

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

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

Observe that this program is the same of the one reported in Example 4 and, therefore, we can use pointers to implement pushdown stores.

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

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

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" [4]. We represent the

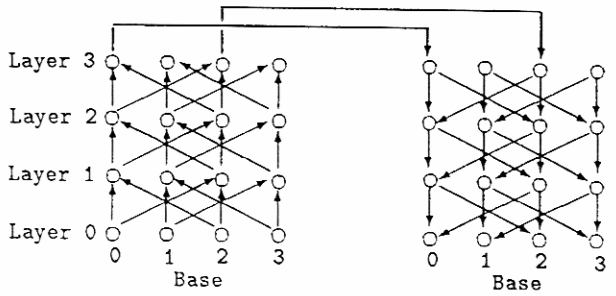

Figure 2: Query graph associated with a cylindric database

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. For each tuple (a, b) 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 $(0 \leq j \leq h\text{-}1)$ denotes the layer and i $(0 \leq i \leq b\text{-}1)$ denotes the position in the layer.

Each node (i, j) in G_A (resp. G_B) with $0 \leq j < h\text{-}1$ (resp. $0 < j \leq h\text{-}1$) has g leaving arcs whereas each node (i, j) in G_A (resp. G_B) with $0 < j \leq h\text{-}1$ (resp. $0 \leq j < h\text{-}1$) has g ending arcs. In particular, for each node (i, j) in G_A such that $j < h\text{-}1$ and for each cardinal k between 0 and $g - 1$ there is an arc in G_A from the node (i, j) to the node $(i{+}k(b \ div \ g), j{+}1)$. Analogously, for each node (i, j) in G_B such that $(j > 0)$ and for each cardinal k between 0 and $g - 1$ there is an arc in G_B from the node (i, j) to the node $(i{+}k(b \ div \ g), j\text{-}1)$.

The sub-graph G_C (used only in the second query) contains $(h \times c)$ arcs, where c $(\leq b)$ denotes the number of arcs connecting nodes in the level j $(0 \leq j \leq h\text{-}1)$ of G_A to 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 query graph with $b = 4$, $h = 4$, $g = 2$ and $c = 2$, where only the arcs of G_C connecting nodes in the layer 3 are reported, is pictured in Figure 2.

For the experiments below reported, 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 G_C (used only in the query Q_2) has 60 arcs. In the 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 3 and 4, respectively.

The prototypes used for the evaluation of queries have been implemented in C++ and they run on personal computer. The results of the experiments have been carryed out by using a PC with CPU Intel 486, 8 Mbyte of Ram and clock of 66 MHz.

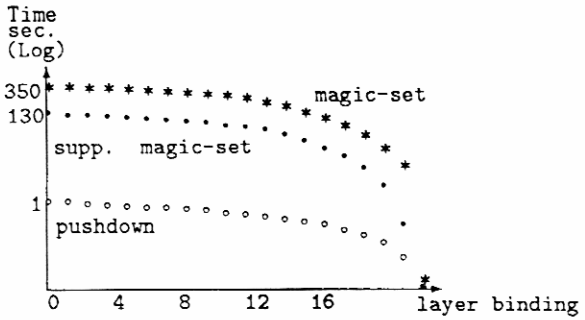

Figure 3: Experimental results for the query Q_1 with $b = 15$, $h = 20$, $g = 3$

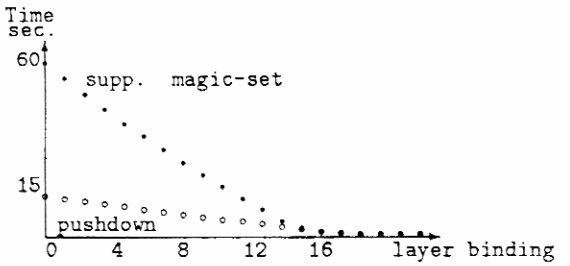

Figure 4: Experimental results for the query Q_2 with $b = 15$, $h = 20$, $g = 3$, $c = 4$

Concluding, we observe that

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. This is quite obvious since a small number of arcs in the query-graph is used for the evaluation of the queries. Moreover, 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 much better than the supplementary magic-set method.

A large number of experimental results using a number of different queries and databases have been performed. The results of these experiments are reported in [11].

References

[1] F. Afrati, S. Cosmadakis. Expressiveness of Restricted Recursive Queries. In *Proc. ACM SIGACT Symp. on Theory of Computing*, 1989, 113–126.

[2] F. Afrati, C.H.. Papadrimitriou. The parallel complexity of simple chain queries. In *Proc. of the Sixth ACM PODS*, 1987, 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 and R. Ramakrisnhan. Performance evaluation of data intensive logic programs. In *Found. of Deductive Datab. and Logic Progr.* (Minker ed.), 1988, 439–518.

[5] C. Beeri, P. Kanellakis, F. Bancilhon, and R. Ramakrisnhan. Bounds on the Propagation of Selection into Logic Programs, *J. Comp. System Sc.*, Vol. 41, No. 2, 1990.

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

[7] G. Dong, On Datalog Linearization of Chain Queries. In J.D. Ullman, editor, *Theoretical Studies in Computer Science*, Academic Press, 1991, 181–206.

[8] G. Dong, Datalog Expressiveness of Chain Queries: Grammar Tools and Characterization. In *Proc. of the Eleventh ACM PODS*, 1992, 81–90.

[9] S. Greco, D Saccà and C. Zaniolo, The Pushdown Method to Optimize Chain Logic Queries. In *Proc. 22nd ICALP Conference*, 1995.

[10] S. Greco and C. Zaniolo, Optimization of linear logic programs using counting methods. In *Proc. of the Extending Database Technology*, 1992, 187–220.

[11] S. Greco, D. Saccà, E. Spadafora and C. Zaniolo, Grammars and Automata to Optimize Chain Logic Queries. In *ISI-CNR Technical Report.* 1996.

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

[13] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[14] J. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.F. Ullman. Argument Reduction by Factoring. In *Proc. of the 15th VLDB Conference*, 1989, 173–182.

[15] J. Naughton, R. Ramakrisnhan, Y. Sagiv, and J.F. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proc. ACM SIGMOD Conference*, 1989, 235–242.

[16] J.E. Hopcroft, and J.F. Ullmann. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[17] 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, 231–278.

[18] R. Ramakrisnhan, Y. Sagiv, J.F. Ullman, and M.Y. Vardi. Logical Query Optimization by Proof-Tree Transformation. In *J. Computer and System Sc.*, No. 47, 222-248, 1993.

[19] J.DD. Ullmann. *Principles of Data and Knowledge-Base Systems.* Volume 1 & 2, Computer Science Press, New York, 1989.

[20] J.D. Ullmann. The Interface Between Language Theory and Database Theory. In *Theoretical Studies in Computer Science* (Ullmann ed.), Academic Press, 1991, 133–151.

[21] J.D. Ullmann and A. Van Gelder. Parallel Complexity of Logical Query Programs. In *Proc. of the 27th IEEE Symp. on Foundations of Computer Science*, 1986, 438–454.

[22] L. Vielle. Recursive Query processing: The Power of Logic. *Theoretical Computer Science.* No. 69, 1989, 1–53.

[23] P.T. Wood. Factoring Augmented Regular Chain Programs. *Proc. of the 16th VLDB Conference.* Brisbane, Australia, 1990, 225–263.

[24] M. Yannakakis. Graph-Theoretic Methods in Database Theory. In *Proc. of the Ninth ACM Symposium on Principles of Database Systems*, 1990, 230–242.