EVALUATING LINEAR-RECURSIVE LOGIC QUERIES
AGAINST CYCLIC RELATIONS

by

Eram Gery and Yoav Ñaz

Technical Report #427
July 1986
EVALUATING LINEAR-RECURSIVE LOGIC QUERIES AGAINST CYCLIC RELATIONS

by

Eran Gery and Yoav Raz
Technion - Israel Institute of Technology
Computer Science Department
Haifa 32000, Israel

ABSTRACT

A top-down algorithm is given to evaluate a simple class of recursive logic-queries. The intended scope of the algorithm is linear recursive queries against cyclic relations. A relation defined by a linear recursive rule can be constructed by evaluating successive join expressions generated by rule expansions according to the resolution principle. This naive method was improved by Henschen and Naqvi, and a more efficient method was presented by Banchilon et al. These algorithms use termination condition based on the acyclicity of the base-relations, i.e. they are not applicable if the database may have cyclic relations. We extend the method for cyclic relations and compare the method to other methods applicable for this domain. The presented algorithm is shown to be advantageous in many cases.
1. INTRODUCTION

Recently, with the increasing interest with deductive databases, few methods for evaluating logic queries denoted by Horn-clauses have been proposed. The main effect of using the clauses is that also non first order queries can be expressed, using recursive Horn-clauses or "recursive queries".

As mentioned in [BR] we can state three heuristic criterions to test recursive query evaluation method.

1. Discarding non relevant facts,
2. Avoiding duplication of work,
3. Using small-arity intermediate relations (e.g a unary relation is "better" than a binary relation).

A class of efficient methods exist to evaluate the linear recursive queries. (A query is linear recursive if every rule has at most one recursive predicate on its right side.) There are basically two kinds of methods to evaluate logic queries: the **top-down** (backward chaining) and the **bottom-up** (forward chaining). Intuitively, to satisfy criterion 1 above, we must use the bindings (the determined attributes) from the query during the evaluation process to discard non relevant facts. This can be done by propagating the constants from the query goal down the tree of deduction. Unfortunately, we cannot use these bindings in a bottom up computation, but the advantage behind the bottom-up strategy is that the convergence of the process can be detected trivially.

The first method to evaluate linear queries presented by Henschen and Naqvi [HN] was a top-down method. This method satisfies criterions 1, 3 above but there are redundant computations. Other two methods presented at [B+] called the **Magic-sets** method and the **Counting** method. The Magic-sets method is a bottom-up evaluation preceded by a phase which "simulates" the binding propagation top-down, and it does not satisfy criterion 3. The Counting method is an improvement of [HN] method by eliminating the redundant deductions. Both, Henschen-Naqvi and Counting methods, are not applicable if the base relations may have cycles, since their termination detection is based on the acyclicity of the relations.

In this paper we extend the top-down method to cyclic relations. In some of the definitions we use the terminology of [SZ]. The idea is not demonstrated for the general case, but rather upon a simple typical case for clarity.
2. SOLVING LOGIC QUERIES TOP-DOWN

To avoid complex notation and definitions we shall describe the method using the a typical linear rule which is:

\[
p(x, y) : A(x, z), p(z, w), B(w, y)
\]

\[
p(x, y) : -C(x, y)
\]

and the query \( p(a, ?) \), where \( a \) stands for some binding of the first argument.

However, this method is applicable for any linear recursive rule.

The predicates \( A, B, C \) are base relations, which means database physical relations, and the predicate \( p \) is a deduced relation, or a view.

The right side of any rule is a logic conjunction between the relations, which can be interpreted as natural join expression between two relations having columns sharing the same variable.

The relation \( p \) can be defined by an infinite set of join expressions, generated by expanding the predicate \( p \) in the recursive rule using repeated substitutions. The expansion is done according the unification principle, and the relation \( p \) is the union of all the productions starting by the query as non-terminal symbol, expanding it, and select the expressions which include only base relations.

Since the relations \( A, B, C \) are finite, and the Horn-clauses do not include function symbols (i.e. new elements cannot be generated), the deduced relation \( p \) is finite, and actually we need to evaluate only a finite set of join expressions.

Example 1: The set of expressions which define the answers to the query:

\[
C(a, ?)
\]

\[
A(a, z)C(z, w)B(w, ?)
\]

\[
A(a, z)A(z, z)C(z, w)B(w, w)B(w, ?)
\]

where "?" expresses the set of answers retrieved from every expression evaluated by projection on the last column after making the appropriate natural-join (as mentioned above) for every expression. The union of these sets, is the answer to the query.

Now we consider two problems:
1. How can we implement efficiently the joins in order to minimize the redundant computations?

2. How can we detect the convergence of the process?

The first problem is considered in [HN], and we have basically adopted the method described there. [HN] considers also the termination detection problem, but the solution suggested has two main drawbacks:

1. It maintains data structure which is exponential in the number of elements in the database.

2. There are cases where the algorithm does not stop.

The algorithm in [HN] is correct over base relations without cycles. In this case, there is a more efficient method, called the Counting method, which is also strongly based on the fact that the database is without cycles. These two methods are very similar, but the latter computes the answers more efficiently, by eliminating some redundant deductions which appear in the algorithm of [HN].

3. STRATEGY

Before presenting the method formally, we summarize the general strategy in the following two principles.

1. To improve the naive way of repeating the join expressions evaluations, we would like to use results from previous evaluations, i.e., we have to keep intermediate results from the previous computation.

2. Termination detection is usually achieved by detecting stabilization of the answers set. As we shall see, this criterion is impractical here because of the way the computation of answers is carried out (top-down); a lack of change in the answers set for several steps of answers computations does not guarantee the stabilization of the set. Instead, we shall observe the stabilization of other sets which we can detect, and the stabilization of all of them together will guarantee the stabilization of the answers set.

This strategy is implemented as follows:

At every evaluation of a join expression:

\[ A(\alpha_1)A(\alpha_2)\ldots A(\alpha_{n-1},\alpha_n)C(\gamma_n)B(\gamma_{n-1})\ldots B(\gamma_1) \]

that we shall denote:
first evaluate the set $x_n$ of all possible values for $x_n$ (the "A part"); Then, evaluate the expression $C(x_n,y_n)B^n(y_n,?)$ to extract the answers; using the set $x_n$ (the "B part"). This has satisfied the two principles:

1. To evaluate $x_{n+1}$ we need the set $x_n$ only, which has been evaluated and kept during the previous step. The way of computing $x_n$ follows principle 1 above.

2. The sets mentioned in principle 2 above, are sets of elements on the paths from elements in $x_n$ to answers defined by the relation $C$ join $B$ i $\leq n$. As we shall see, these sets have the desired property, and in each step we can know what elements of $x_n$ are not useful, i.e. can not derive new answers.

4. DEFINITIONS AND BASIC PROPERTIES

In the general case, the Magic-Graph is defined by the linear recursive rule and by the query (see [SZ]). In this paper we define the Magic-graph according to the typical rule and the query discussed before.

Definition 1: The Magic-Graph

The Magic-graph $G(V,E)$ is a directed graph defined for the query $p(a,?)$ and the rule $p(x,y):=A(x,z),p(z,w),B(w,y)$ as follows:

i. $a \in V$

ii. if $u \in V$ and $A(u,v)$ then $v \in V$ and $(u,v) \in E$.

(In fact, the relation $E$ can be viewed as a sub-relation of $A$).

Definition 2: The Counting-sets

The Counting-sets $M_i$ $i=1,2,...$ are defined with respect to the Magic-graph $G$ as follows:

$M_0 = \{a\} \quad M_i = \{v \mid \text{there is a path in } G \text{ (possibly, cyclic) from } a \text{ to } v \text{ with the length } i \}$.

The Counting-sets cover the nodes of the Magic-graph but are not necessarily distinct. Obviously, the number of Counting-sets is infinite iff the Magic-graph is cyclic.
Lemma 1:
The set of answers to the query, \(\text{Answer}\), can be expressed using the Counting-sets as follows:

\[
\text{Answer} = \bigcup_{i \in \mathbb{N}} \{x \mid (\exists v)(\exists u) (M_i(v), C(v,u), B^i(u,x))\} \bigcup \{x \mid C(a,x)\}
\]
(Where \(M_i(v)\) stands for \(v \in M_i\))

where \(B^i\) is defined:

\[
B^1(x,y) = \{(x,y) \mid B(x,y)\}
\]
\[
B^i(x,y) = \{(x,y) \mid (\exists u) B^{i-1}(x,u) B(u,y)\}.
\]

The proof follows directly from the definitions of \(M_i\), \(B^i\) and the answers of the query. \(\square\)

Corollary:
If the Magic-graph does not have cycles, Lemma 1 induces an algorithm to evaluate the query.

It follows from the fact that the number of non empty Counting-sets \(M_i\) is finite. (In fact, in this case, the most efficient algorithm is the Counting algorithm \([B+], [SZ]\), but the algorithm which simply implements the above comments is that described in \([HN]\).)

Definition 3: The Cyclic-Closure of the Magic-Graph

\(C(G) = \{v \mid v \text{ belongs to an infinite set of Counting-sets}\}\)

The Cyclic-Closure \(C(G)\) can be computed in \(O(|E|)\) time, by the following algorithm:

1. Let \(G(V,E)\) be the Magic-graph:
2. and assume the nodes in \(V\) are marked with serial numbers 1, 2, ..., \(|V|\):
3. Construct \(D\), an array of size \(|V|\), where \(D(v)\) is the input degree of node \(v\):
4. Construct \(S\) be array of sets where \(S(i)\) is the set of all the nodes with input degree \(i\):
5. while not empty \((S(0))\) do
   1. extract element from \(S(0)\):
   2. \(S(0) \leftarrow S(0) - \{v\}\)
   3. for every element \(u\) where \((v,u) \in E\) do
      1. \(D(u) \leftarrow D(u) - 1\)
      2. move \(u\) from \(S(D(u)+1)\) to \(S(D(u))\);
   end for;
end while;
6. \(C(G) := \bigcup_{i \geq 0} S(i)\)
Lemma 2:
The algorithm above computes \( C(G) \).

Proof:
First we notice the simple fact that since \( G \) is finite, an equivalent definition of \( C(G) \) is
\[
C(G) = \{ v \mid \text{there is a cyclic path from } a \text{ to } v \}
\]

*note*: A cyclic path is a path in which there is a node which occurs more than once.

Now we can prove the following:

1. The algorithm can not remove from \( S(0) \) elements which belong to the closure.
   This comes from the property that an element \( v \) can get into \( S(0) \) only if all nodes which have a path to \( v \) have been in \( S(0) \). This property is an invariant of the algorithm, and is proved by induction on the iteration number.
   Now, suppose that an element on a cycle is removed. Then by the previous property, it must have been removed before, and we have a contradiction. After we have shown that elements on cycles can not be removed, we can use this property again to deduce that their predecessors cannot be removed.

2. All the elements which are not in the closure get into \( S(0) \) during the algorithm.
   This property is proved by induction on the length of the longest path from \( a \) to every such an element.

\(\square\)

Lemma 3:
The complexity of the algorithm above is \( O(\mid E \mid) \).

Proof:
Step 1 can be done by scanning \( E \) in \( O(\mid E \mid) \).
Step 2 can be done in \( O(\mid V \mid) \) using \( D \) computed at step 2.
Step 3 uses every edge once, so it can be computed in \( O(E) \). \(\square\)
Lemma 4:
There exist $j$ such that $M_j \subseteq C(G)$.

Proof: Since $G$ is finite, obviously $j < |V| - 1$ the maximal simple path length in a graph. $\square$

Lemma 5:
Let $j$ be the minimal integer such that $M_j \subseteq C(G)$, then for every $k > j$, $M_k \subseteq C(G)$.

Proof: Suppose that there exists a node $t \in M_k$, $k > j$ and $t \notin C(G)$. By definition there is a path of length $k$ from $a$ to $t$. Let $v$ be the $j$'th node of this path, then by definition $v \in M_j$, and there is a path from $v$ to $t$. But $v \in C(G)$, i.e. it belongs to an infinite number of Counting-sets, so also $t$. A contradiction. $\square$

Definition 4: The sets $Ans_i(v)$.
Let $v \in V$ (an element from the Magic graph). The sets $Ans_i(v)$ $i=1, 2, \ldots$ are defined as follows:

$Ans_i(v) = \{x \mid (\exists y) C(v, y) B^i(y, x)\}$

Note: If $v \in M_i$ then the elements of $Ans_i(v)$ are answers to the query; if $v \notin M_i$ then the elements of $Ans_i(v)$ are not necessarily answers.

Lemma 6:
Let $j$ be the minimal integer such that $Ans_j(v) \subseteq \bigcup_{n < j} Ans_n(v)$.

1. There is always such finite $j$.
2. For every $k > j$, $Ans_k(v) \subseteq \bigcup_{n < j} Ans_n(v)$. or, in other words, $v$ is not useful for deriving any more answers.

(all its answers have been found till step $j - 1$).

Proof:
1. This holds since the base relation $B$ is finite.
2. The proof comes from the basic fact that:

$Ans_{i+1}(v) = f(Ans_i(v))$

where $f(S)$ is the set defined by $\{x \mid (\exists y \in S) B(y, x)\}$.
And \( f \) has the following two properties:

(i) \( f(S_1 \cup S_2) = f(S_1 \cup S_2) \) which implies:

(ii) if \( S_1 \subseteq S_2 \) then \( f(S_1) \subseteq f(S_2) \).

Let \( \text{Ans}_j(v) \) be as stated in the lemma. Now using the above properties and induction on \( i \), we get:

\[
\text{Ans}_{j+i}(v) \subseteq \bigcup_{j' \leq j} \text{Ans}_{j'}(v).
\]

\[\square\]

Definition 5: The predicates \( \text{Useful}_i \) \((i \neq 1, 2, \ldots)\).

Let \( v \) be an element of the Magic -Graph. Let \( j \) be the one defined in Lemma 6. Then

\[\text{Useful}_i(v) = \begin{cases} \text{true} & \text{if } i < j \\ \text{false} & \text{otherwise} \end{cases}\]

Theorem 1:

Let \( i \) be the maximal index such that \((M_i(v), C(v, u), B(u, ?))\) still derives new answers for the query. Then \( i = \min(j, k) \) where:

1. \( j \) is the minimal integer such that \((\forall v \in V)(\text{Useful}_i(v) = \text{false})\)

2. \( k \) is the minimal integer such that \((\forall v \in C(G))(\text{Useful}_k(v) = \text{false} \text{ and } M_k \subseteq C(G))\).

Proof:

By Lemma 1, the set of answers to the query, \( p(a, ?) \) is created from the evaluations of the "B part" (i.e. \( C(v, y)B^i(y, ?) \)) using elements \( v \) from the Counting-sets. If condition 1 above holds then all these elements are no more useful, i.e. they will not derive new answers by Lemma 6. Hence, if we detect condition 1 we can stop the evaluations. If condition 2 holds then by Lemma 4, the second part of the condition assures that for all the elements \( v \in M_i \), the condition \( i > k \) implies \( v \in C(G) \). The first part of the condition assures us that all these elements are not-useful any more by Lemma 6 again. \( \square \)

5. THE QUERY EVALUATION ALGORITHM

The algorithm is based on the theorem above. The Counting-sets are computed iteratively while for every
element in the Magic -Graph its usefulness is evaluated. The sets and relations mentioned before are imple-
mented in the algorithm and have similar names. However, the implementation requires some more data
structures which are defined within the algorithm's comments.

1. compute \( G(V,E) \);  
   /* The Magic -Graph */
2. compute the set \( C(G) \);  
   /* The Cyclic -closure */
3. \( \text{not-useful} \leftarrow \emptyset \);  
   /* \( \text{not-useful} \) accumulates \( v \) such that \( \text{useful}(v) = \text{false} \) */
   \( M \leftarrow \{a\} \);  
   /* \( M \) is the current Counting -set */
   for every \( v \in V \) do
      \( \text{Useful}(v) \leftarrow \text{true} \);  
      \( \text{Last-count}(v) \leftarrow 0 \);  
      /* \( \text{Last-count}(v) \) is the index of the last Counting -set which contains \( v \) */
      end for;
4. while not ((\( \text{not-useful} = V \)) or (\( M \subseteq C(G) \) and \( \text{not-useful} \supseteq C(G) \))) do
   \( M \leftarrow \pi_2(\{a \in M \mid A\}) \);  
   if \( M = \emptyset \) then stop;
   for every \( v \in M \) do
      if \( \text{Last-count}(v) = 0 \) then
         \( \text{Ans}(v) \leftarrow \pi_2(\{a \in M \mid C\}) \);  
         /* \( \text{Ans}(v) \) stands for \( \text{Ans}_s(v) \) */
         \( \text{UAns}(v) \leftarrow \text{UAns}(v) \cup \text{Ans}(v) \);  
         /* \( \text{UAns}(v) \) is the union of all the previous \( \text{Ans}(v) \) */
      end if
      while \( \text{Last-count}(v) < i \) and \( \text{Useful}(v) \) do
         \( \text{Last-count}(v) \leftarrow \text{Last-count}(v) + 1 \);
         \( \text{Ans}(v) \leftarrow \pi_2(\{a \in M \mid B\}) \);  
         if \( \text{UAns}(v) \supseteq \text{Ans}(v) \) then
            \( \text{not-useful} \leftarrow \text{not-useful} \cup \{v\} \);
            \( \text{Useful}(v) \leftarrow \text{false} \);
            else
            \( \text{UAns}(v) \leftarrow \text{UAns}(v) \cup \text{Ans}(v) \);
         end if
      end while;
      if \( \text{Useful}(v) \) then
         \( \text{Answer} \leftarrow \text{Answer} \cup \text{Ans}(v) \);
      end if
   end for;
   \( i \leftarrow i + 1 \);
end while;

At the end the set \( \text{Answer} \) contains all the answers of the query.

Note: \( \pi_2(\{a \in M \mid C\}) \) stands for \( \{x \mid C(y,x) \wedge y \in M\} \)

Theorem 2:

The above algorithm is complete and sound with respect to the linear rule and every database relations
\( A, B, C \).

Proof: The algorithm implements the computations described and justified in the lemmas and theorems.
The *Magic-sets* and similarly the *Magic-counting* methods evaluate the following tuples of $p$ (and some more):

$$p(u_i, v_j) \quad i=1,2,\ldots,n \quad j=1,2,\ldots,n$$

Hence it is of size $O(n^2)$. For this example our method takes $O(n)$ time.

**Remark:** since $b \in C(G)$ the algorithm uses condition 2 of theorem 1 when it terminates. The element $b$ demonstrates why we need to identify the elements which belong to $C(G)$.

**ACKNOWLEDGEMENT**

We would like to thank Shimon Even for suggesting the idea for the $O(|E|)$ cyclic-closure algorithm.

**REFERENCES**


6. COMPARISONS WITH OTHER METHODS

The preferred method for linear recursive query so far, that is correct with respect to cyclic relations, is the Magic-sets method \([B+1]\). This method evaluates the query bottom-up but considers only the relevant tuples from \(A\), i.e. the Magic-Graph edges. In other words, it restricts the relation \(A\) before making the bottom-up joins. There is another relevant method called Magic-Counting \([SZ]\) which integrates the Magic-sets method and the Counting method, but as the Magic-Graph becomes "more cyclic", the method reduces to Magic-sets.

The drawback of the Magic-sets method is that it evaluates the answers for the whole Magic-Graph (for each \(v \in V\) \(p(v,?)\) is computed), where our method evaluates the answers only for the query constant "\(a\)". The Magic-sets method requires to maintain a binary relation in its bottom-up evaluations (for our typical case), while our method uses a unary relation, the current Counting-set. This affects the amount of computation as shown in example 2.

Example 2:
The following graph in fig. 1 represents the relations \(A, B, C\), and for this database state our method is shown to be more efficient than the Magic-sets and Magic-counting methods. The rules and the query are those discussed in the paper.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig1.png}
\caption{Fig. 1}
\end{figure}

The \(A\) relation's tuples are represented by arcs of the type \(\longrightarrow\), \(B\) by \(--\rightarrow\), and \(C\) by \(--\rightarrow\).