ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

Speeding up the Counting Method by Computing Heritage Functions in Topological Order

Yangjun Chen

Published in collaboration with the
British Computer Society

Speeding up the Counting Method by Computing Heritage Functions in Topological Order

Yangjun Chen

Department of Computer Science, TU Chemnitz-Zwickau, 09107 Chemnitz, Germany

Abstract

In this paper, an optimal method for evaluating linear recursive datalog queries is proposed. The method is based on the concepts of so-called heritage appearance function and heritage selection function. By computing such functions in topological order, a counting-like strategy can be implemented, which requires only linear time for non-cyclic data.

1 Introduction

Deductive databases generalize relational databases by including not only base predicates (or relations), but also derived predicates (or views). A derived predicate is defined by means of one or more deductive rules.

A lot of strategies for processing deductive rules, especially for recursively defined rules have been proposed (see [1-15, 17-23]). In this paper, we confine ourselves to the counting method [4, 20] for linear recursion and try to improve its performance in the case of non-cyclic data. This method seeks to perform a compile-time transformation of the database, based on the query form, into an equivalent form which enables a bottom-up computation to focus on relevant tuples. As with the magic set method [3, 4], the transformed programs consist of two rule sets: *counting rules* and *modified rules*. Thus, the computation can be done in a two-phase approach. In the first phase, we produce a counting set by evaluating the counting rules. In the second phase, we produce all answers by evaluating modified rules with the counting set being used to restrict the computation. According to the graphic analysis performed in [17, 18], the worst-case time complexity of this method is $O(n \cdot e)$, better than magic sets. Here, we introduce two new concepts: *heritage appearance function* and *heritage selection function*, and transform many algebraic operations into simple computations of such functions (i.e., some boolean operations) in topological order. In this way, high efficiency can be obtained not only due to the simplicity of boolean operations, but also due to the elimination of much redundancy by using binary sequence (string of 1's and 0's) property.

The paper is organized as follows. In the next section, we introduce the concepts of linear recursive queries, query graph and query dependency graph, and describe the counting method in a graphical formalism. In Section 3, we define the *heritage appearance function* and *heritage selection function*, and present an optimal evaluation algorithm for *canonical strongly linear recursion* (CSLRs), which reduces the cost of both the first and second phase of the counting method to $O(e)$. In Section 4, we discuss more complicated linear recursion, i.e., *non-interdependent recursion* (NILR) and *interdependent recursion* (ILR). In Section 5 and 6, we prove the correctness of the refined algorithm and compare the time complexity of the refined algorithm with other well-known strategies. Section 7 is a short conclusion.

2 Basic Concepts and Counting Method

In this section, we define some concepts and terminologies which are necessary for introducing our efficient method.

2.1 Basic Definitions

The language of a deductive database consists of the variables, constants, and predicate names in the data base. We adopt some informal notational conventions for them. Variables will normally be denoted by the letters u, v, x, y and z (possibly subscripted). Constants will normally be denoted by the letters a, b and c (possibly subscripted). Predicate names will normally be denoted by the letters p, q, r , and s (possibly sub scripted). In the absence of function symbols, a term is either a constant or a variable. Occasionally, it will be convenient not to apply these conventions rigorously. In such a case, possible confusion will be avoided by the context. An atom is an n -ary predicate, $p(t_1, t_2, \dots, t_n)$, $n \geq 0$, where p is a predicate name and $t_1, t_2, \dots, t_n$ are terms. A literal is an atom or the negative of an atom. A positive literal is just an atom. A negative literal is the negation of an atom. A rule is a first-order formula of the form

$$q \leftarrow p_1, p_2, \dots, p_m, \quad m \geq 0.$$

q is called the head and the conjunction $p_1, p_2, \dots, p_m$ is called the body of the rule. Each p_i is a body literal. When $m = 0$, the rule is of the form

$$q \leftarrow$$

and is known as a *unit clause*.

An atom $p(t_1, t_2, \dots, t_n)$, $n \geq 0$ is *ground* when all of its terms $t_1, t_2, \dots, t_n$, are constants. A *ground rule* is one in which each atom in the rule is ground. A *fact* is a ground unit clause. The definition of a predicate p is the set of rules which have p as the head predicate. A *base predicate* is defined solely by facts. The set of facts in the database is also known as the *extensional database*. A rule that is not a fact is known as a *derivation rule*. A *derived predicate* is a predicate which is defined solely by derivation rules. A derived (base) literal is one whose predicate is derived (base). The set of derivation rules is also known as the *intensional database* or program.

In addition, in this paper, we use the term graph to refer to the directed graph, since we don't discuss the undirected ones at all. We assume that a graph G is specified as follows: for each node v_i in the graph, there is a set of *successors* $\text{adj}(v_i) = \{v_j \mid (v_i, v_j) \text{ is an edge of } G\}$. Without loss of generality, we assume that G has no self-loops, i.e., for all nodes v_i , $v_i \notin \text{adj}(v_i)$. For an edge (v_i, v_j) , node v_i is called the *source* or *tail* and node v_j is called the *destination* or *head* of the edge. We denote the transitive closure of a graph G by G^* . The successors of v_i in G^* are the *descendants* of v_i in G . The *strongly connected component* (or *strong component*) of node v_i is defined as $V_i = \{v_i\} \cup \{v_j \mid (v_i, v_j) \in G^* \text{ and } (v_j, v_i) \in G^*\}$. The component V_i is not trivial if $V_i \neq \{v_i\}$. As will be seen, our algorithms is based on depth-first traversal of graphs, so we review now some relevant definitions on it. Depth-first traversal induces a spanning tree on a graph based on the order in which nodes are visited. If we assume that the main routine in depth-first traversal is $\text{visit}(v_i)$ for a node i , then there is an edge (v_i, v_j) in the spanning tree, if there is a call to $\text{visit}(v_j)$ during the execution of the call $\text{visit}(v_i)$. An edge (v_i, v_j) in the graph G is called a *tree edge*, if it belongs in the spanning tree. An edge (v_i, v_j) in the graph G but not in the spanning tree is called a *forward edge*, a *back edge*, or a *cross edge*, if in the panning tree, v_j is a descendant of v_i , v_j is an ancestor of v_i , or v_j is not related to v_i with an ancestor-descendant relationship, respectively. For every strong component, its node r on which $\text{visit}(r)$ is first called is the root of the strong component.

2.2 Query Graph and Query Dependency Graph

We distinguish among three kinds of linear recursions: *canonical strongly linear recursion* (CSLR), *non-interdependent linear recursion* (NILR) and *interdependent linear recursion* (ILR). A CSLR is a program contains only one linear recursive rule besides the non-recursive rules. For example, the following program is a typical CSLR.

- (1) $rp(x, y) : - flat(x, y)$
- (2) $rp(x, y) : - up(x, z), rp(z, w), down(w, y).$

In contrast, an NILR may contain any many recursive predicates but no interdependency happens. That is, each recursive predicate in an NILR does not appear in the body of any rule defining the other. As an example, consider the following abstract program:

- $$\begin{aligned}
 q(x, y) &: - s_1(x, y), \\
 q(x, y) &: - s_2(x, y), \\
 s_1(x, y) &: - r_1(x, y), \\
 s_1(x, y) &: - p_1(x, z), s_1(z, w), q_1(w, y), \\
 s_2(x, y) &: - r_2(x, y), \\
 s_2(x, y) &: - p_2(x, z), s_2(z, w), q_2(w, y),
 \end{aligned}$$

in which recursive predicates s_1 and s_2 are non-interdependent. The ILR is the most complicated linear recursion, where some recursive predicates may be interdependent. That is, one recursive predicate may appear in the body of a rule defining another. For example, in the following program, recursive predicate q_1 is dependent on q_2 while q_2 itself depends on both q_3 and q_4 .

- $$\begin{aligned}
 q_1(x, y) &: - p_1(x, z), r_1(z, y), \\
 q_1(x, y) &: - p_1(x, z), q_1(z, w), q_2(w, u), q_3(u, y), \\
 q_1(x, y) &: - p_1(x, z), q_1(z, w), q_2(w, u), q_3(u, y), \\
 q_2(x, y) &: - p_2(x, z), q_3(z, w), q_2(w, u), q_4(u, y), \\
 q_3(x, y) &: - r_3(x, y), \\
 q_3(x, y) &: - p_3(x, z), r_3(z, w), q_3(w, y), \\
 q_4(x, y) &: - p_4(x, y), \\
 q_4(x, y) &: - p_4(x, z), q_4(z, y).
 \end{aligned}$$

In order to investigate the behavior of a CSLR evaluation, we associate a directed graph (called query graph) with each query against a CSLR program. A query graph basically consists of three parts: up-part (UP), flat-part (FP), and down-part (DP). The UP is that relation part which is reachable from the constants in the query. The FP is that part which can be reached using the non-recursive rule, and the DP can be reached using the recursive rule. For example, if the *up*, *flat*, and *down* predicates in the above CSLR program are defined as:

- $$\begin{aligned}
 up &= \{ (a_1, a_2), (a_1, a_3), (a_1, a_4), (a_1, a_5), (a_2, a_3), \\
 &\quad (a_2, a_4), (a_2, a_5), (a_3, a_4), (a_3, a_5), (a_4, a_5) \} \\
 flat &= \{ (a_5, b_2), \\
 down &= \{ (b_5, b_4), (b_4, b_3), (b_3, b_2), (b_2, b_1), \},
 \end{aligned}$$

then the query graph representing the query $? - rp(a_1, y)$ will be as shown in Figure 1, where the edges going up represent tuples in UP, the broken edges represent tuples in FP, and the edges going down represent tuples in DP.

We will use the notation $G_u = \langle N_u, E_u \rangle$, $G_f = \langle N_f, E_f \rangle$ and $G_d = \langle N_d, E_d \rangle$ to represent the subgraphs induced from: UP, FP, and DP, respectively. In Figure 1, node $a_1 \in N_u$ represents the constant in the query $rp(a_1, y)$ and we call such a node the *source node*. Given a subgraph G (i.e., G can be G_u , G_f , or G_d) and a subset X of G , we denote by $adjG(X)$ ($adj^{-1}G(X)$) the set of all nodes v_j such that the edge (v_i, v_j) ((v_j, v_i)) is in G and v_i is in X . In other words, $adjG(X)$ is the set of all nodes that are adjacent to some node in X ,

Figure 1: Graph representing input relations

whereas $\text{adj}^{-1}G(X)$ is the set of all nodes having at least one adjacent node in X . It is easy to see that these two sets of nodes can be computed using the rules stated above, which can be also expressed as relation-algebra expressions.

As an example, consider the above query graph. Say that

$$X = \{a_4, a_5\}.$$

Then

$$\text{adj}G_u(X) = \{a_5\},$$

$$\text{adj}^{-1}G_u(X) = \{a_1, a_2, a_3, a_4\},$$

and

$$\text{adj}G_f(X) = \{b_5\}.$$

In order to specify the program decomposition strategy with respect to NILRs and ILRs, we define a graph, called *query dependency graph*, for each linear recursive program as follows. The nodes of the graph represent the recursive predicates appearing in the program. An edge $p \rightarrow q$ connects p and q iff p appears in the body of some rule defining q and is labelled with an *adornment* that is expected for p in terms of the current query submitted to the system. An adornment for an m -ary predicate $p(t_1, t_2, \dots, t_m)$ is a string of length m made up of the letters b and f , where b stands for *bound* and f stands for *free*. We obtain an adornment for a predicate as follows. During a computation, each argument t_i , $1 \leq i \leq m$, of the literal $p(t_1, t_2, \dots, t_m)$ is expected to be bound or free, depending on the information flow. If t_i is expected to be bound (free), it acquires a b (f) annotation, and so the length of the adornment string is m . For example, for the ILR program shown above, we have a query dependency graph as shown in Figure 2 if the query against this program is of the form: $?- q_1(c, y)$.

Figure 2: Illustration of query dependency graph

Obviously, the query dependency graph for an NILR is just a node set, containing no edges at all; and the query dependency graph for an ILR is a graph containing no cycles (otherwise, it is non-linear recursive.)

In fact, a query dependency graph corresponds exactly to a composition of the corresponding program. That is, each node p in the graph can be thought of as a CSLR defining p with the other recursive predicates (appearing in it) handled as non-recursive ones. These CSLRs are connected to each other with the adornment

labelling edges. More importantly, in terms of such a graph, each CSLR can be compiled (as discussed in Section 3) independently but only once. Of course, there may be many different methods for partitioning a program. For our purpose, however, decomposing a program into a set of CSLRs is desired so that the optimization ideas proposed in Section 3 can be directly employed for a decomposed program.

2.3 Counting Method

Since we are going to refer frequently to the counting method, we feel that it will be convenient to describe it in a graphical formalism [18]. Figure 2 represents the basic idea of the counting method (essentially, it corresponds to the implementation of the supplementary counting method [4].) It works as follows. Let $U_i (i \geq 0)$ contain all nodes v in G_u that have distance i from the source node s . In the first phase, the method computes U_i (notice that, in general, such sets are not disjoint and are called the counting set. This process corresponds to the evaluation of the counting rules.) Suppose that U_g contains the nodes with the greatest distance (thus g) from s . In the second phase, we start computing the set D_g of all nodes in G_f that are adjacent to some nodes of U_g in G_f . Then we compute D_{g-1} as the set of all nodes in G_d that are adjacent to some node of U_{g-1} in G_f and that are adjacent to some node of D_g in G_d . We continue until we compute D_0 , which contains all the answers (answer nodes) of the query (this process corresponds to the evaluation of the modified rules.) If the graph G_u is cyclic, this version of the counting method is not safe.

```

 $U_0 := \{s\}; i := 0;$ 
while  $U_i \neq \emptyset$  do
  begin
     $U_{i+1} := \text{adj}_{G_u}(U_i); i := i + 1;$ 
  end
   $D_{i-1} := \text{adj}_{G_f}(U_{i-1});$ 
for  $j := i - 1$  downto 1 do
     $D_{j-1} := \text{adj}_{G_f}(U_{j-1}) \cup \text{adj}_{G_d}(D_j);$ 
Answer  $:= D_0.$ 
    
```

Figure 3. Counting method

From the description of the algorithm, we can see that in the case of acyclic data the first loop can be performed $O(|N_u|)$ times and every iteration has cost $O(|E_u|)$. Therefore, the total cost of the first phase is $O(|N_u| \cdot |E_u|)$. Similarly, the second phase has a total cost of $O(|N_u| \cdot |E_d|)$. Hence, the cost of the counting method for acyclic queries is $O(n \cdot e)$, where n and e denote the number of nodes and edges, respectively, in the graph representing the input relations.

The purpose of this paper is to reduce the time complexity of both first and second phases using a new method based on the notions of heritage appearance function and heritage selection function. In the following, we discuss these two functions and the corresponding computation methods in Section 3.

3 Optimal Algorithm for CSLRs

In this section, we describe our efficient algorithm for CSLRs. First, in Subsection 3.1, we discuss the optimization for CSLRs without cyclic data in detail. Then, we sketch a method for CSLRs with cyclic data briefly in Subsection 3.2

3.1 CSLRs without Cyclic Data

Similar to the counting method, our algorithm works in a two-phase manner. In the first phase, we compute the appearance function sequences for each node of G_u in $O(e)$ time. Then, in the second phase, we extract the

answers in terms of such appearance function sequence in some way. As will be seen later, the time requirement of the second phase is also bounded by $O(e)$.

3.1.1 The First Phase of the Algorithm

Here we describe the first phase of the counting method. First, we present the concept of appearance function which was introduced in [1] to describe the possible distances of a node v in G_u from a source node s .

Definition 1 The appearance function $A_{v,s}(i)$ of a node v with respect to a source node s is a binary valued function:

For all integer $i \geq 0$:

$$A_{v,s}(i) = \begin{cases} 1 & \text{if there exists a path from } s \text{ to } v \\ & \text{length } i, \\ 0 & \text{otherwise.} \end{cases}$$

Then the sequence $A_{v,s} = A_{v,s}(0), A_{v,s}(1), A_{v,s}(2), \dots$ corresponds to the different appearances of v with different distances from s . For example, for the query graph shown in Figure 1, we have

$$\begin{aligned} A_{a_5,a_1} &= 01111, \\ A_{a_4,a_1} &= 01110, \\ A_{a_3,a_1} &= 01100, \\ A_{a_2,a_1} &= 01000, \\ A_{a_1,a_1} &= 10000. \end{aligned}$$

An observation shows that in the case of non-cyclic data, if the height of G_u is h (we define the height of an acyclic graph to be the number of the nodes on the longest path in the graph), then the length of each $A_{v,s}$ is bounded by h and $h \leq |N_u|$. Thus, the number of 1's appearing in all appearance function sequences with respect to G_u (denoted as N_{1-bit}) is bounded by $h \cdot |N_u|$. If we can find a method to generate each "1" only once, the time complexity of the first phase will be reduced to $O(h \cdot |N_u|)$. In fact, we can find an algorithm which can generate all such sequences in linear time.

In order to generate each "1" only once, we introduce another concept, so called heritage appearance function for the nodes of G_u , which can be defined as follows.

Definition 2 The heritage appearance function $HA_{v',s}^v(i)$ of a node v' with respect to a node v (and a source node s) is a binary valued function:

$$HA_{v',s}^v(i) = \begin{cases} 1 & \text{if } A_{v,s}(i-1) = 1 \text{ and } (v, v') \in E_u, \\ 0 & \text{otherwise.} \end{cases}$$

Then $HA_{v',s}^v = HA_{v',s}^v(1), HA_{v',s}^v(2), \dots$ can be easily computed by shifting $A_{v,s}$ right 1 bit and filling the emptied position with 0. Further, we will use $HA_{v,s}$ to denote the sequence

$$HA_{v,s}^{v_1} \cup HA_{v,s}^{v_2} \cup \dots \cup HA_{v,s}^{v_j},$$

where $v_1, v_2, \dots, v_j$ are the direct precedents of v in G_u and $\cup$ is used to denote OR operation on the corresponding positions of two binary sequences respectively.

This definition hints an efficient computation method, in which all appearance function sequences can be generated in linear time. First, we have the following proposition.

Proposition 1 Let s be the source node of a G_u . Then, for all nodes v of G_u except s , we have

$$A_{v,s} = HA_{v,s} = HA_{v,s}^{v_1} \cup HA_{v,s}^{v_2} \cup \dots \cup HA_{v,s}^{v_j},$$

where $v_1, v_2, \dots, v_j$ are the direct precedents of v in G_u .

proof. It follows directly from the definitions of $A_{v,s}$ and $HA_{v,s}$.

Based on the above proposition, we propose an algorithm which works in a two-step manner and can generate all appearance function sequences for the nodes of G_u in linear time. In the first step, we produce a directed graph corresponding to G_u . In the second step, we first find a topological order (for G_u) with the property that all precedents of a node n_i are before n_i in the order. Then we compute the appearance function sequence for each node in such an order as follows. At the beginning, the appearance function sequence of the source node $A_{s,s}$ is initialized to "100 ... 0". Then, the second node in the topological order can be obtained by shifting $A_{s,s}$ right 1 bit, filling the emptied position with 0. The i th node can be computed using the equation given in Proposition 1. Since the appearance function sequences of all direct precedents of a node have already been generated before the node is encountered, we can evaluate the appearance function sequence for each node of G_u in a topological order without difficulty. More importantly, in this way, each $A_{v,s}$ can be generated by only performing d_v shifting operations and d_v logical OR operations, where d_v represents the indegree of v . That is, to compute the appearance function sequence for a node v , we need only to shift each ($i = 1, \dots, d_v$) right 1 bit and then to add them together, where v_i ($i = 1, \dots, d_v$) are the direct precedents of v .

This process can be summarised as follows. (In the following algorithm, the operation $ShiftRight(A_{v,s})$ shifts $A_{v,s}$ right 1 bit, filling the emptied position with 0.)

```

procedure first – phase
  begin
    (1) generate  $G_u$ ;
    (2) find a topological order for  $G_u$ ;
    let  $\{v_1, v_2, \dots, v_n\}$  be the corresponding topological order and  $v_1 = s$  be the source node;
     $A_{v_1,s} := "10... 0"$ ;
    for  $i = 2$  to  $n$  do
      let  $v_{i_1}, \dots, v_{i_j} \in \{v_1, v_2, \dots, v_{i-1}\}$  be the direct precedents of  $v_i$ 
      for  $j = 1$  to  $l$  do
         $HA_{v_i,s}^{v_{i_j}} := ShiftRight(A_{v_{i_j},s});$ 
       $A_{v_i,s} := HA_{v_i,s}^{v_{i_1}} \cup \dots \cup HA_{v_i,s}^{v_{i_l}};$ 
    end
  
```

Obviously, the time complexity of the above algorithm is $O(|E_u|)$. On the one hand, G_u can be generated in $O(|E_u|)$ time and the topological order for it can also be found in $O(|E_u|)$ (see [16]). On the other hand, the cost of generating an $A_{v,s}$ is bounded by $O(d_v)$ and then the total cost of generating all appearance function sequences is

$$\sum_{v \in \{v_1, \dots, v_n\}} O(d_v) = O(|E_u|).$$

In practice, the entire time spent for doing the shifting operations and the OR operations for a node can be taken to be $O(1)$. Therefore, the total cost of generating all appearance function sequences should be $O(N_u)$.

3.1.2 The Second Phase of the Algorithm

In terms of the appearance function sequences, the answers can be extracted by determining the distances for the nodes of G_d (the graph induced from the down-part DP). First, we define the heritage selection functions.

Then we discuss how such functions can be used to find the correct answers in linear time.

In [1], a concept: *selection function* is introduced to provide a termination condition for the counting method in the case of cyclic data. Based on this notion, we propose a new concept: *heritage selection function* for each node $e \in G_d$ to select those nodes which belong to the answer set and are reachable from e . In this way, any repeated access to an edge in G_d can be replaced by a simple boolean computation. In what follows, we first give the definition of the selection function. Then we define the notion of the heritage selection function and derive a refined algorithm based on the computation of the function for each node of G_d in a topological order.

In terms of the definition of the appearance functions, we know that if $v \in N_u, w \in N_d, s$ is the source node, and $(v, e) \in E_f$, then w is in the answer set iff

$$A_{v,s}(i) = A_{w,e}(i) = 1 \quad \text{for some } i \quad (1)$$

Since the flat relation is many-to-many in general, we define the following function to represent the union of all the appearance functions of those nodes $v \in N_u$ such that $(v, e) \in E_f$.

Definition 3 The selection function $S_e(i)$ of a node e in G_d is a binary valued function:

For all integer $i \geq 0$:

$$S_e(i) = \begin{cases} 1 & \text{if } A_{v,s}(i)=1 \text{ for some } v \in N_u \\ & \text{and } (v, e) \in E_f, \\ 0 & \text{otherwise.} \end{cases}$$

Then the sequence $S_e = S_e(0), S_e(1), S_e(2), \dots$ indicates the distances from e with the property that if a node in G_d possesses one of such distances from e , it should be in the answer set. This property is called the *selection property*. For example, if we have $(u, e), (v, e) \in E_f$ and $A_{u,s} = 01010, A_{v,s} = 00101$, where s represents the source node in G_u , then $S_e = A_{u,s} \cup A_{v,s} = 01111$, which indicates that the nodes (in G_d) with the distance one, two, three or four from e are answer nodes.

From the definition of the selection function, one can rewrite condition (1) as follows: A node $w \in N_d$ is in the answer set iff for some $e \in N_f \cap N_d$,

$$S_e(i) = A_{w,e}(i) = 1 \quad \text{for some } i \quad (2)$$

Now consider a sequence $S_{e'}$ obtained by shifting S_e left one bit and filling the emptied position with 0. Such a sequence has an important meaning that if there exists a node v such that $(e, v) \in E_d$, then $S_{e'}$ indicates the distances of some nodes from v which belong to the answer set if they are in G_d . This property, together with the fact that the answer node can be checked dynamically in terms of selection function sequences, provides a possibility to optimize the evaluation of the second phase.

Definition 4 The heritage selection function $H_{e'}^e(i)$ of a node e' (in G_d) with respect to a node e is a binary valued function. For all integer $i \geq 0$:

If e has no precedents in G_d , then

$$H_{e'}^e(i) = \begin{cases} 1 & \text{if } S_e(i+1) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

In other cases,

$$H_{e'}^e(i) = \begin{cases} 1 & \text{if } S_e(i+1) = 1 \text{ or } H_{e_1}^{e_1}(i+1) = 1 \text{ for} \\ & \text{some } e_1 \text{ such that } (e_1, e) \in N_d, \\ 0 & \text{otherwise.} \end{cases}$$

Then the sequence $H_{e'}^e = H_{e'}^e(0), H_{e'}^e(1), H_{e'}^e(2), \dots$ indicates the distances from e' , which have the selection property.

We will use H_e to denote the sequence $H_e^{e_1} \cup H_e^{e_2} \cup \dots H_e^{e_l}$, where $e_1, e_2, \dots, e_l$ are the direct precedents of e in G_d . Intuitively, H_e can be used to propagate control information to extract answer subsets correctly.

Definition 5 The generalized selection function $GH_e(i)$ is a binary valued function. For all integer $i \geq 0$:

$$GH_e(i) = \begin{cases} 1 & \text{if } S_e(i) = 1 \text{ or } H_{e_1}^{e_1}(i) = 1 \text{ for some } e_1 \\ & \text{such that } (e_1, e) \in N_d, \\ 0 & \text{otherwise.} \end{cases}$$

In terms of the above definitions, we have the following proposition.

Proposition 2 If (e, e') is an edge in E_d and $GH_e(k) = 1$ for some $k \geq 1$, then we have

$$GH_{e'}(k-1) = 1$$

proof. It follows directly from the definitions of $S_e(k)$ and $GH_e(k)$.

From the above discussion, we know that GH_e indicates the distances of all those nodes from e , which belong to the answer set if they are in G_d . For each $e \in N_f \cap N_d$, we can produce its S_e by evaluating the (modified) non-recursive rule with the corresponding appearance function sequences being propagated (see Definition 3). Obviously, this can be done in linear time (see [1] for a more detailed description). Similar to the treatment of the first phase, we then generate a graph corresponding G_d and compute all generalized selection function sequences in a topological order, thereby generating the answers dynamically in terms of the generalized selection function sequences computed so far. In the following algorithm, the operation $ShiftLeft(GH_e)$ returns a binary sequence obtained by shifting GH_e left 1 bit and filling the emptied position with 0.

procedure *second – phase*

```

begin
(1) generate  $S_e$  for each  $e \in N_f \cap N_d$ ;
(2) generate  $G_d$ ;
(3) find a topological order for  $G_d$ ;
    let  $\{e_1, e_2, \dots, e_n\}$  be the corresponding topological order
    for  $i = 1$  to  $n$  do
        let  $e_{i_1}, \dots, e_{i_l} \in \{e_1, e_2, \dots, e_{i-1}\}$  be the direct
        precedents of  $e_i$ 
        for  $j = 1$  to  $l$  do
             $H_j := ShiftLeft(GH_{e_{i_j}})$ ;
             $GH_{e_i} := S_{e_i} \cup H_1 \dots \cup H_j$ ;
        * if  $GH_{e_i}(0) = 1$  then
            insert  $e_i$  into the answer set;
    end

```

Note the statement marked with *. By executing this statement, each node is checked whether it belongs to the answer set in terms of its generalized selection function sequence. That is, if the 0th position of the generalized selection function sequence of a node e is "1", then e should be in the answer set (and then e is called an answer node). It is because in this case either (s, e) is in E_f , where s is the source node, or there is a path of length $2i + 1$, $i > 0$, from the source node s to e such that the first i edges are in E_u , the $(i+1)$ th edge

is in E_f , and the last i edges are in E_d . (see Proposition 4 given in the next section.)

Example 1 Continue with our running program. Suppose that the facts in the database can be represented as a graph shown in Figure 4.

Figure 4: Graph representing input relations

Given the query $? - rp(a_1, y)$, the algorithm *first - phase* first generates G_u , which corresponds to the *up* part of the graph shown in Figure 4. Then the topological order for it can be found in linear time:

$$a_1 \longrightarrow a_2 \longrightarrow \dots a_{n-1} \longrightarrow a_n.$$

In this order, the algorithm will produce the following appearance function sequences:

$$\begin{aligned} A_{a_1, a_1} &= "10 \dots 00", \\ A_{a_2, a_1} &= ShiftRight(A_{a_1, a_1}) = "010 \dots 00", \\ A_{a_3, a_1} &= ShiftRight(A_{a_1, a_1}) \cup ShiftRight(A_{a_2, a_1}) \\ &= "0110 \dots 00", \\ &\dots \dots \\ A_{a_n, a_1} &= ShiftRight(A_{a_1, a_1}) \cup ShiftRight(A_{a_2, a_1}) \cup \dots \\ &\quad \cup ShiftRight(A_{a_{n-1}, a_1}) \\ &= "011 \dots 1". \end{aligned}$$

For this example, the algorithm *second - phase* will produce only one selection function sequence:

$$S_{b_n} = "011 \dots 1".$$

The selection function sequence for each S_{b_j} ($j = 1, \dots, n - 1$) can be taken to be $"00 \dots 0"$. Then, *second - phase* will generate a graph which corresponds to the *down* part of the graph shown in Figure 4. The topological order for it is:

$$b_n \longrightarrow b_{n-1} \longrightarrow \dots b_2 \longrightarrow b_1.$$

In this order, the generalized selection function sequences will be generated and in terms of them the answer set will be produced dynamically:

$GH:$	Answer set:
$GH_{b_n} = S_{b_n} = "011 \dots 1"$	$\emptyset$
$GH_{b_{n-1}} = S_{b_{n-1}} \cup ShiftLeft(GH_{b_n})$	$\{b_{n-1}\}$
$\quad = "11 \dots 10"$	
$GH_{b_{n-2}} = S_{b_{n-2}} \cup ShiftLeft(GH_{b_{n-1}})$	
$\quad \cup ShiftLeft(GH_{b_n})$	
$\quad = "11 \dots 10"$	$\{b_{n-1}, b_{n-2}\}$

$$\begin{aligned}
 \dots \dots GH_{b_1} &= S_{b_1} \cup ShiftLeft(GH_{b_2}) \cup \dots \\
 &\quad \cup ShiftLeft(GH_{b_n}) \\
 &= "11\dots 10" \quad \{b_{n-1}, b_{n-2}, \dots, b_1\}.
 \end{aligned}$$

3.2 About Cyclic Data

The evaluation of recursive queries in the case of cyclic data is intrinsically a difficult problem. There has been considerable effort directed toward this issue in recent years and several efficient methods have been proposed, which improve the performance to $O(n \cdot e)$. Here, we try to combine the technique discussed in Section 3 with some ideas developed in the previous research and show a new method which requires only $O(e + n^2)$ time to produce all answers to a query.

In terms of the method proposed in [1], the appearance function sequence of a node appearing in cycles can be described with two subsequences: a transient sequence and a steady sequence. That is, for some node v its $A_{v,s}$ will be represented as a regular expression of the form $\alpha\beta^*$ over the alphabet set 0, 1, where α stands for the transient part and β stands for the steady part. For example, the appearance function sequence of the node v_4 in the graph shown in Figure 5(a) is of the form: $000(100)^*$. Then $\alpha = "000"$ and $\beta = "100"$. When a node appears in more than one cycles, it becomes harder to determine α and β without doing duplicate work. For example, the appearance function sequence of v in the graph shown in Figure 5(b) is $00101(1)^*$, which can be obtained by performing a depth-first traversal of the graph or by a simple computation using the formula given in [1].

Figure 5: Cyclic data

An important property of such a sequence is that if the number of nodes of a graph (possibly containing cycles) is n , then the length of α , denoted $|\alpha|$, is less than n^2 . The proof of this property can be found in [1], and a similar proof can be found in [22]. Based on this property, a method is presented by [13], which requires $O(n \cdot e)$ time to compute α and β .

To reduce the time complexity, we associate each node with a mark bit, which is initialed and changed as follows. First, we determine the topological order for G_u by ignoring back edges of it. Then, we compute the appearance function sequence along the topological order iteratively. At the beginning, we set each mark bit to be "0". (The appearance function sequence for each node is also assigned "0".) Whenever a node is encountered during a scan of the topological order, we calculate the new appearance function sequence for it in terms of the equation given in Definition 1. If the value is the same as the old one or the length of the new appearance function sequence is larger than n^2 , we set its mark bit to be "0", otherwise to "1". Therefore, when we compute a new appearance function sequence for a node, we do not consider those precedants of it, whose mark bit is "0", because no contribution to the new sequence value for the current node will be made by them. Obviously, much work will be saved in this way. In addition, it provides a sufficient and necessary condition for termination when all the mark bits become "0".

4 Optimal Algorithms for NILRs and ILRs

In general, the idea described above can not be directly employed to handle more complex linear recursive programs. However, we can always partition the rules of a program into several CSLRs so that the optimization idea discussed in the previous section can be directly exploited. In the following, we address this problem in detail.

First, we consider NILRs, which can simply be decomposed. For example, to evaluate a query like $?-q(c, y)$ against the NILR program given in 2.2, we compute $?-s_1(c, y)$ and $?-s_2(c, y)$ separately, which will be derived from the original query. Obviously, the technique developed in 3.1 can directly be utilized for each $?-s_i(c, y)$ ($i = 1, 2$). Accordingly, we have the following algorithm.

```
procedure evaluation – for – NILR(query)
begin
  for each NILR  $q_i$  involved in the evaluation of query do
    call first – phase;
    call second – phase;
end
```

In terms of the query dependency graph, we know that a recursive algorithm should be implemented to evaluate an ILR. Let $q_1, \dots, q_i, \dots, q_m$ be recursive predicates involved. We can associate each q_i with a set of query graphs of the form: $G^i = G_u^i \cup G_f^i \cup G_d^i$ with each for a query of the form $?-q_i(c_i, y)$, where c_i represents a constant appearing the query generated with respect to q_i during an evaluation and can be dynamically determined by the constant propagation. Then, for each query graph, the technique developed for CSLR can be utilized since each $q_i(c_i, y)$ is evaluated essentially against a canonical strongly linear recursive program. To this end, we change *first – phase* and *second – phase* a bit so that recursive calls can be implemented. In fact, the corresponding procedures have the following form:

first – phase – ILR(c, q),
second – phase – ILR(q).

Their formal descriptions are given below.

```
procedure first – phase – ILR
begin
  {generate  $G_u$ ;
   at each moment (during the generation of  $G_u$ )
   a recursive predicate  $q_i$  is encountered, do the
   following operations:
   construct  $?-q_i(c_i, y)$ , where  $c_i$  represents a constant
   determined by the constant propagation;
   call first – phase – ILR(ci, qi);
   call second – phase – ILR(qi);}
  find a topological order for  $G_u$ ;
  let  $\{v_1, v_2, \dots, v_n\}$  be the corresponding topological
  order and  $v_1 = s$  be the source node;
   $A_{v_1, s} := "10... 0"$ ;
  for  $i = 2$  to  $n$  do
    let  $v_{i_1}, \dots, v_{i_j} \in \{v_1, v_2, \dots, v_{i-1}\}$  be the direct
    precedents of  $v_i$ 
    for  $j = 1$  to  $l$  do
       $HA_{v_i, s}^{v_{i_j}} := ShiftRight(A_{v_{i_j}, s});$ 
```

```


$$A_{v_i,s} := HA_{v_i,s}^{v_{i_1}} \cup \dots \cup HA_{v_i,s}^{v_{i_l}};$$

end

procedure second – phase – ILR
begin
    generate  $S_e$  for each  $e \in N_f \cap N_d$ ;
    {generate  $G_d$ ;
     at each moment (during the generation of  $G_u$ )
     a recursive predicate  $q_i$  is encountered, do the
     following operations:
     construct  $?-q_i(c_i, y)$ , where  $c_i$  represents a constant
     determined by the constant propagation;
     call first – phase – ILR( $c_i, q_i$ );
     call second – phase – ILR( $q_i$ );}
    find a topological order for  $G_d$ ;
    let  $\{e_1, e_2, \dots, e_n\}$  be the corresponding topological
    order
    for  $i = 1$  to  $n$  do
        let  $e_{i_1}, \dots, e_{i_l} \in \{e_1, e_2, \dots, e_{i-1}\}$  be the direct
        precedents of  $e_i$ 
        for  $j = 1$  to  $l$  do
             $H_j := ShiftLeft(GH_{e_{i_j}})$ ;
             $GH_{e_i} := S_{e_i} \cup H_1 \dots \cup H_j$ ;
        if  $GH_{e_i}(0) = 1$  then
            insert  $e_i$  into the answer set;
    end

```

5 Correctness of the Refined Algorithm

In this section, we prove the correctness of our algorithm in the case of non-cyclic data. To this end, we first show that the first phase of our algorithm will generate all appearance function sequences correctly. Then we prove that the second phase can produce all answer nodes.

Proposition 3 In the case of non-cyclic data, the algorithm *first – phase* will generate all appearance function sequences for the nodes of G_u .

proof. We prove the proposition by induction over the generated appearance function sequences on the number of the nodes of G_u .

Basis: If G_u contains only one node (the source node), its appearance function sequence is "10...0" and thus is correct.

Induction step: Suppose that the proposition holds for all graphs containing k nodes ($k \leq n - 1$) and that G_u contains n nodes. Let $v_1 \rightarrow v_2 \dots \rightarrow v_n$ be a topological order for G_u . Then the subgraph of G_u generated by $\{v_1, v_2, \dots, v_{n-1}\}$ contains $n - 1$ nodes. By the induction hypothesis, the appearance function sequence for each $v_i \in v_1, v_2, \dots, v_{n-1}$ can be correctly generated. Let $v_{i(1)}, \dots, v_{i(l)}$ be the direct precedents of v_n in G_u . Then

$$A_{v_n,s} = ShiftRight(A_{v_{i(1)},s}) \cup \dots \cup ShiftRight(A_{v_{i(l)},s}),$$

which is exactly evaluated by the algorithm *first – phase*. Since each $A_{i(j),s}$ ($j = 1, \dots, l$) is correctly evaluated, $A_{v_n,s}$ is correctly evaluated. This completes the proof.

In order to prove the second phase of our algorithm, we first give the following proposition.

Proposition 4 If $e \in N_d$ belongs to the answer set, then there exists a path of length $2i + 1$, $i \geq 0$, from the source node s to e such that the first i edges are in E_u , the $(i + 1)$ th edge is in E_f , and the last i edges are in E_d (such a path is called an answer path.)

proof. See the appendix of [18].

Based on this proposition, we can immediately prove the correctness of the algorithm *second – phase*.

Proposition 5 In the case of non-cyclic data, the algorithm *second – phase* will produce all answer nodes.

proof. First, we claim that the algorithm *second – phase* can correctly produce all generalized selection function sequences for the nodes of G_d . This can be proved in a similar way that Proposition 3 is manifested. Then, we clarify that for each e in the "answer set" produced by *second – phase* there exists an answer path. Consider the statement marked with * in *second – phase*, by which we insert a node into the answer set if the 0th position of its generalized selection function sequence is "1". We distinguish between two cases in which the 0th position of e 's GH is "1". The first case is that there is an edge (s, e) in E_f , where s is the source node, then e is in the answer set and the length of the associated path is $2 \cdot 0 + 1 = 1$. The second case is that among the direct precedents of e there is at least one node e_1 , the 1th position of its GH is "1". If $S_{e_1}(1) = 1$, then there is an answer path of length $2 \cdot 1 + 1 = 3$. Otherwise, consider the direct precedents of e_1 . Similarly, among these nodes there must exist at least another node e_2 , the 2th position of its GH is "1". If $S_{e_2}(2) = 1$, then there is such an answer path of length $2 \cdot 2 + 1 = 5$. Otherwise, we further consider its direct precedents. In this way, we can always find a sequence $e_1, e_2, \dots, e_i$ connecting e_i and e with $S_{e_i}(i) = 1$. Thus, there is a path (of length $2i + 1$, $i \geq 0$), which connects the source node and e . Therefore, in terms of Proposition 4, we know that the answer set is correctly evaluated.

6 Comparison with Other Strategies

In the analysis below, we consider only the following abstract linear recursive program:

$$\begin{aligned} s(x, y) &: - r(x, y) \\ s(x, y) &: - p(x, z), s(z, w), q(w, y). \end{aligned}$$

Assume that the graph representing the relation for "r" contains n_r nodes and e_r edges, the graph for "p" contains n_p nodes and e_p edges, and the graph for "q" contains n_q nodes and e_q edges.

As stated above, in the case of non-cyclic data, the first and second phase of the refined algorithm requires $O(n_p + e_p)$ time and $O(n_q + e_q)$ time, respectively. Therefore, it is a linear time algorithm. In contrast, the counting method requires $O(n_p \cdot e_p)$ time in the first phase and $O(n_p \cdot e_q)$ time in the second phase. The magic-set method is another bottom-up algorithm which works also in two phases. The first phase of it consists of determining all nodes in N_u (N_u is then called the magic set). In the second phase, the method computes all possible pairs of nodes (i, j) satisfying the following conditions:

- (1) i is in N_u , j is in N_d ,
- (2) there is another pair (i', j') which is in E_f or is produced in previous steps, and
- (3) $(i, i') \in E_u$ and $(j, j') \in E_d$.

Therefore, if we use the seminaive approach, the cost of the first phase of the magic-set method is $O(e_p)$. The cost of the second phase is

$$\sum_{(i,j) \in A} indegree(i) \times outdegree(j) = O(e_p \cdot e_q),$$

where A denotes the set of answer tuples. The graph shown in Figure 6 helps to clarify this result.

Figure 6: Illustration for time complexity analysis

From this graph, we see that crossing an answer tuple, say (i, j) , each edge incident to j will be visited $indegree(i)$ times by the magic set method. Since the number of answer tuples is bounded by $(n_p \cdot n_q)$, the cost of the magic set method is $O(e_p \cdot e_q)$ [17, 18]. In recent years, there has been considerable effort directed toward the extension of the counting method for dealing with cyclic relations, such as the level-cycle merging method proposed by [14, 22], the synchronized counting method [1] and the method proposed by Haddad and Naughton [13]. All those methods try to reduce the time complexity to $O(n \cdot e)$ in the case of cyclic data. But no progress has been made in the direction of decreasing the time complexity of the counting method itself. In addition, a lot of experiments have been done [5] and show that QSQR, a well-known top-down strategy [21], has the same time complexity as the magic set method. At an abstract level, the expansion phase of QSQR can be viewed as two processes: a constant propagation process and a variable instantiation process. The former corresponds to the traversal of the graph for "p". The latter corresponds to the traversal of the graphs for "r" and "q". Therefore, the analysis for the magic set method applies to QSQR.

7 Conclusion

In this paper, two new concepts: heritage appearance function and heritage selection function have been introduced and an efficient algorithm for evaluating recursive queries have been developed. Based on the computation of such functions in topological order, this algorithm reduces the cost of the counting method significantly and can be used to treat with non-cyclic relations. The algorithm is efficient not only due to the simplicity of boolean operations, but also due to the elimination of much redundancy by using binary sequence property. In the case of non-cyclic data, the algorithm requires only $O(n + e)$ time, where n and e denotes the number of nodes and edges, respectively, in the graph representing the input relations.

References

- 1 Aly, H. and Ozsoyoglu, Z. M., Synchronized Counting Method, *Proc. of the 5th International Conf. on Data Engineering*, Los Angeles, 1989.
- 2 Balbin, G. S. Port, K. Ramamohanarao and K. Meenakshi, Efficient Bottom-up Computation of Queries on Stratified Databases, *J. Logic Programming*, November 1991, 295-344.
- 3 F. Bancilhon, D. Maier, Y. Sagiv and J. D. Ullman, Magic Sets and Other Strange Ways to Implement Logic Programs, *Proc. 5th ACM Symp. Principles of Database Systems*, Cambridge, MA, March 1986, 1-15.

- 4 C. Beeri and R. Ramakrishnan, On the Power of Magic, in: *Int. Journal of Logic Programming*, 1991:10:255-299.
- 5 S. Ceri, G. Gottlob and L. Tanca, *Logic Programming and Databases*, Springer-Verlag, Berlin, 1990.
- 6 Y. Chen and T. Haerder, Improving RQA/FQI Recursive Query Algorithm, in: *Proceedings ISMM - First Int. Conf. on Information and Knowledge Management*, Baltimore, Maryland, USA: ACM, Nov. 1992.
- 7 Y. Chen, A Bottom-up Query Evaluation Method for Stratified Databases, in: *Proceedings of 9th International Conference on Data Engineering*, Vienna, Austria: IEEE, April 1993, pp. 568-575.
- 8 Y. Chen and T. Haerder, On the Optimal Top-down Evaluation of Recursive Queries, in: *Proc. of 5th Int. Conf. on Database and Expert Systems Applications*, Greece, Athens: Springer Verlag, Sept. 1994, pp. 47-56.
- 9 Y. Chen and T. Haerder, An Optimal Graph Traversal Algorithm for Evaluating Linear Binary-chain Programs, in: *Proc. CIKM'94 - the 3th Int. Conf. on Information and Knowledge Management*, Gaithersburg, Maryland: ACM, Nov. 1994, pp. 34-41.
- 10 Y. Chen, On the Bottom-up Evaluation of Recursive Queries, *Int. Journal of Intelligent Systems*, Vol. 11, 807-832 (1996).
- 11 Y. Chen, Magic Sets and Stratified Databases, *Int. Journal of Intelligent Systems*, Vol. 12, 203-231 (1997).
- 12 Y. Chen, Magis Sets Revisited, to appear in *Journal of Computer Science and Technology*, China, April 1997.
- 13 R. W. Haddad and J. F. Naughton, Counting Method for Cyclic Relations, in: *Proceedings of the 7th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986, pp. 16-23.
- 14 J. Han and L.J. Henschen, The Level-Cycle Merging Method, *Proc. of the 1st International Conf. on Deductive and Object-oriented Databases*, Kyoto, 1989.
- 15 J. Han, Chain-Split Evaluation in Deductive Databases, *IEEE Transaction on Knowledge and Data Engineering*, Vol 7, No. 2, April 1995, pp. 261-273.
- 16 D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley Series in Computer Science and Information Processing, 1968, 257-265.
- 17 A. Marchetti-Spaccamela, A. Pelaggi, and D. Sacca, Worst Case Complexity Analysis of Methods for Logic Query Implementation, *Proc. of ACM-PODS*, 87.
- 18 A. Marchetti-Spaccamela, A. Pelaggi, and D. Sacca, Comparison of Methods for Logic-query Implementation, *J. Logic Programming*, 1991:10:333-360.
- 19 R. Ramakrishnan, D. Srivastava and S. Sudarshan, Rule Ordering in Bottom-up Fixpoint Evaluation of Logic Programs, *IEEE Transaction on Knowledge and Data Engineering*, Vol 6, No. 4, Aug. 1994, pp. 501-517.
- 20 K. Wang and L.Y. Yuan, First-Order Logic Characterization of Program Properties, *IEEE Transaction on Knowledge and Data Engineering*, Vol 6, No. 4, Aug. 1994, pp. 518-532.
- 21 D. Sacca and C. Zaniolo, Magic Counting Method, *Proc. ACM-SIGMOD*, May 1987.
- 22 L. Vieille, From QSQ to QoSQ: Global Optimization of Recursive Queries, *Proc. 2th Int. Conf. on Expert Database System*, L.Kerschberg ed., Charleston, 1988.
- 23 C. Wu and L.J. Henschen, Answering Linear Recursive Queries in Cyclic Databases, *Proc. of the 1988 International Conf. on Fifth Gen. Computer Systems*, Tokyo, 1988.