Synchronized Counting Method

Hussien Aly* and Z. Meral Ozsoyoglu†

Computer Engineering and Science Department
Case Western Reserve University
Cleveland, Ohio 44106

ABSTRACT
The counting method is a simple and efficient method to process linear recursive datalog queries. However, it is only applicable to cases where the relations are acyclic. A direct extension to the counting method is presented which can deal efficiently with both cases of acyclic and cyclic relations. Worst case analysis shows that \( n^2 \) semi-join operations are required where \( n \) is the number of nodes in the graph representing the relevant part of the input relations. The algorithm is shown to be sound and complete, and a comparison with other methods in the literature is included.

1. Introduction
In this paper, we address the problem of efficient evaluation of linear recursive queries in a deductive database in the presence of cyclic relations. In the literature, it has been reported that counting method outperforms most of the other methods in the general acyclic case [BR, SZ, SPS] but it fails when cyclic relations are involved. Although there were many attempts to generalize the counting method, these attempts resulted in expensive and/or more complex algorithms.

We used a Petri Net model, called PNLP [AOz] to simulate and study the behavior of the counting method for cyclic and acyclic cases. Studying the flow of tokens in the PNLP model provides us with a simple way to extend the applicability of the counting method to the cyclic cases. We will call the resulting algorithm the Synchronized Counting Method or "SCM" for short.

Although the SCM is first derived using a Petri Net model, we will present it using a graph-theoretic approach since it is more suitable to prove the correctness of the algorithm. Due to space considerations, the Petri Net model will not be described in this paper. For the details concerning this Petri Net model, we refer the reader to [A], [AAS], and [AOz].

In Section 2, the basic definitions and terminology will be given. We define the data connection graphs corresponding to the logic queries and their relation to the efficiency of the evaluation methods. In Sections 3 and 4, we discuss some of the properties of such graphs and the specific problems that arise when they are cyclic. In Section 5, we present the Synchronized Counting Method. The complexity analysis and the comparison between the proposed method and other related ones are discussed in Section 6. Finally we give our conclusion in Section 7.

2. Basic Concepts and Terminology
We assume that the reader is familiar with the deductive databases terminology. In our presentation, we will use the PNLP model of the counting method and the method of [HN] to develop our algorithm. This model is basically consists of three parts: Determined Part (DP), Exit Part (EP), and Induced Part (IP). Briefly, The DP is that relation part which is reachable from the constants in the query. The EP is that part which can be reached using the exit rule (non-recursive rule), and the IP can be reached using the recursive rule. These parts can be recognized using the query format and the Datalog rules involved [AOz] [HN], as illustrated by the following example.

Example 1:

\[
\text{sg}(X, Y) :- \text{up}(X, Z), \text{sg}(Z, Z1), \text{down}(Z1, Y).
\]

\[
\text{sg}(X, Y) :- \text{flat}(X, Y).
\]

\[
\text{sg}(a, ?) ?
\]

Then we can calculate DP = up(X, Y), EP = flat(X, Y), and IP = down(X, Y).

We will use a graph representation similar to that described in [BMSU], [SPS], and others. These graphs are called Data Connection Graphs or simply...
"DCGs". DCGs represent the three relations DP, EP and IP. For example, if the up, flat, and down relations are defined as:

up = \{(a,b),(b,c),(b,d),(c,b),(c,e),(f,e)\}
flat = \{(a,b),(d,i),(e,j)\}
down = \{(i,m),(i,p),(j,i),(k,j),(l,j),(l,m),
       (m,n),(n,l),(n,o),(p,h)\}

then the data connection graph representing the query in example 1 will be as in Figure 1, where thin arcs represent tuples in DP, broken arcs represent tuples in EP, and thick arcs represent tuples in IP.

![Figure 1](image)

In this paper, we assume only linear rules that can be reduced to a form like the above example and we will use the notation \((N_D, E_D), (N_E, E_E),\) and \((N_I, E_I)\) to represent the sets of (nodes, edges) of the DP, EP, and IP respectively. We may also use the notation \(v \in D_P\) when there is no confusion, to denote \(v \in N_D\). Similar notations are used for EP and IP. In Figure 1, node \(a \in D_P\) represents the constant in the query \(sg(a, ?)\) and we call such a node the source node. Note that not all the tuples in the relations are represented in the DCGs. Instead, the DCGs represent only the tuples that can be reached from the source node. Throughout the paper, when we mention DP, EP, or IP we mean only the parts represented in the DCGs. Nodes like \(h, i,\) and \(j\) which are in \(N_R \cap N_I\) are called exit drivers.

Given a query to a deductive database, the DCGs represent the minimum complete search space for the tuples that satisfy this query. The efficiency of an evaluation method depends on whether or not it uses these DCGs (or equivalent), and how it searches for the answers in these graphs.

For example, Naive and Semi-Naive methods do not use the DCGs, in fact, they use the whole relations. But Semi-Naive, during the search, never uses the same tuple twice which make it more efficient than the Naive one. Magic Sets method uses the DCG of the DP instead of the whole relation, and uses the same search strategy as the Semi-Naive method. In [HN], the method uses the DCGs corresponding to the query with a search strategy that groups the nodes in the DP according to their distances from the source node, and then searches the DCG of the IP to that specified distances. This scheme optimizes the search in the DP, but, as mentioned in [HN], not the search in the IP. With some more storage, the counting method is able to optimize the search in the IP also.

Although the search strategies for [HN] method and counting method made them efficient, they are unsafe when cyclic relations are considered. Although many attempts has been made to generalize these methods, there is still room for an efficient method for dealing with the recursive queries in the general case. Since we are going to refer frequently to the counting method, we feel that it will be convenient to describe it briefly. Figure 2 represents the basic idea of the counting method where \(C_S_i\) represents the counting set obtained at iteration \(i\). In this Figure, and throughout the paper, the symbol \(\bowtie\) denotes a semi-join operation followed by a projection on the other attribute. In the next section, we will explore some of the properties of the DCGs when they contain cycles.

```plaintext
begin
  /* Phase I */
  i = 0; CS_0 = \{a\};
  repeat begin
    i = i + 1; CS_i = CS_{i-1} \bowtie\bowtie DP end;
    until CS_i = \varnothing; 
    for j = i - 1 down to 0 do ECS_j = CS_j \bowtie\bowtie EP; 

  /* Phase II */
    successors = ECS_{i-1}; 
    for j = i - 1 down to 1 do
     successors = (successors \bowtie\bowtie IP) \cup ECS_{j-1};
    Answers = successors; 
end.

Figure 2

3. Properties of Cyclic Data Connection Graphs

The problems arise from cyclic relations when evaluating recursive queries mainly because of the difficulty to identify a simple termination condition that guarantee completeness and finiteness of the
method. With the counting method in mind, we can relate these problems to the processing of the DP and IP subgraphs.

For a cyclic DP, the question is how to decide whether a given distance of a node from the source node (query node) is because of cycles or not; and if it is because of cycles, then the future distances of this node need to be determined. In other words, the cyclic behavior of a node in the DP needs to be identified. If the IP is also cyclic, then there will be the question of when we can say that we already collected all the answers, i.e. the question of the completeness of the method.

Another problem which is related to the way the counting method searches the IP in the second phase, is to determine the starting distance of this phase. Counting method, for acyclic relations, starts the second phase by considering only the nodes in DP which have the largest distance from the source node (i.e. last counting set), and then includes the previous sets in the following iterations. In case of a cyclic DP, the largest distance is not defined. In this section and the next one, we will show how these different problems can be solved while preserving the simplicity of the counting method.

A DCG is a connected digraph in which all the nodes are reachable from the source node. We will use the concept of appearance functions* to describe the possible distances of a node d from the source node s. The appearance function $A_{d,s}(n)$ of a node d and a source node s is a binary valued function defined as follows:

For all integers $n \geq 0$,

$$ A_{d,s}(n) = \begin{cases} 
1 & \text{if there exist a path from s to d of length } n, \\
0 & \text{otherwise} 
\end{cases} $$

We will use $A_{d,s}$ to denote the sequence $A_{d,s}(0) A_{d,s}(1) A_{d,s}(2) \cdots$. For example, $A_{d,s} = 00101\ldots$ means that $A_{d,s}(0) = 0$, $A_{d,s}(1) = 0$, $A_{d,s}(2) = 1$, and so on.

For acyclic graphs, there exists $m \geq 0$ for which $A_{d,s}(n) = 0$ for all $n > m$. So, we can describe the appearance functions of the nodes in these graphs by finite expressions. Unfortunately, this is not true for cyclic graphs. However, the next proposition shows that in the cyclic cases, the appearance functions are periodic and can be described by finite expressions. The proof of the proposition, can be found in [AOz1].

**Proposition 1:** Let C be a strongly connected digraph and d is a node in C. Let $p_1$, $p_2$, ... , $p_j$ be the length of all simple directed circuits in C, and let $g = \text{gcd}(p_1, \ldots, p_j)$. Then there exist integer $m \geq 0$ such that:

1. $m = 0 \pmod{g}$,
2. $A_{d,s}(m) = 1$,
3. For all integers $n > m$; $A_{d,s}(n) = 1$ if $n = 0 \pmod{g}$, and
4. $m < V^2$ where $V$ is the number of nodes in C.

Using this proposition, we can describe the appearance functions to be consisting of two parts: The first part corresponds to $n < m$, and the other corresponds to $n \geq m$. We will borrow the terminology of transient part (for $n < m$) and steady state part (for $n \geq m$) from the control theory. To simplify the following discussion, we will represent $A_{x,y}$ as a regular expression of the form $\alpha\beta^*$ over the alphabet set $\{0,1\}$, where $\alpha$ is the transient part and $\beta$ is the steady state part. For example, $A_{s,y} = 01001(01)^*$. For a specific value of $n$, $A_{x,y}(n)$ will return the $n^{th}$ position of this expression.

As the reader might have noticed, the above proposition as stated has two restrictions:

1. The node and the source node are the same, i.e. the proposition is applicable only to $A_{x,y}$, where $x=y$.
2. The graph is a single strongly connected component.

However, removing these two restrictions does not change the fact that the appearance functions are periodic, and each can be described by a finite expression of length $< V^2$. The values of $\alpha$ and $\beta$ are affected by removing these restrictions as follows:

1. For $A_{x,y}$ where $x \neq y$, there are, in general, many different paths (acyclic) from y to x. This may result in a larger (and different) transient part in order to wait for the longest of such paths. Also the period of the steady state part may be shorter since the cycles may be initiated by different

---

*This name comes from the fact that it describes when tokens are placed in specific places in the PNLP model.

+ $\text{gcd}(x, y, z)$ is the greatest common divisor of $x$, $y$, $z$. 
paths.

(2) If we consider a general data connection graph which may contain several strongly connected components, then \(A_{x,y}\) is periodic iff there is a path from \(y\) to \(x\) which includes some node in a cycle. If some paths have nodes in more than one component, then the period of \(A_{x,y}\) will be less than or equal to the period of any of these components.

Using this proposition, we are able to identify the cyclic behavior of a node in the DCGs. Also we will use it to define the complexity of the Synchronized counting Method (SCM) as will be discussed later.

4. Selection Functions and the Synchronization Process

In this section, we will present solutions to the problems associated with the processing of the IP as mentioned in the beginning of the previous section. In particular, we will develop a necessary and sufficient condition for some nodes in the IP to be in the answer set using properties of the appearance functions of the nodes in IP only.

Using the graph interpretation of logical queries (See for example [BMSUJ, [BR], [SPS]), if \(U \in \mathcal{N}D, v \in \mathcal{N}I, s\) is the source node, and \((u,e) \in \mathcal{E}P\) then \(v\) is in the answer set iff

\[
A_{d,e}(k) = A_{d,e}(k) = 1 \quad \text{for some } k \geq 0. \quad \cdots \quad (1)
\]

Since the exit relation \(\mathcal{E}P\) is many-to-many in general, let us define the new function \(S_e(x)\). For \(e \in \mathcal{N}I\) \(S_e(x)\) is defined as the union of all the appearance functions of all nodes \(d \in \mathcal{N}D\) such that \((d,e) \in \mathcal{E}P\). We will call \(S_e(x)\) the selection function of the exit driver \(e\) since it will be used to select the nodes in IP which belong to the answer set derived from \(e\).

More formally:

\[
S_e(x) = \begin{cases} 
1 & \text{if } A_{d,e}(n) = 1 \text{ for some } d \in \mathcal{N}D \\
& \text{and } (d,e) \in \mathcal{E}P, \\
0 & \text{otherwise}
\end{cases}
\]

Example 2: Let \((d_1,e), (d_2,e) \in \mathcal{E}P, A_{d_1,e} = 01011\) and \(A_{d_2,e} = 01101(10)^*\), then we can write the sequence describing the function \(S_e(x)\) as:

\[
S_e = 01111(10)^*.
\]

From the definition of the selection functions, one can directly rewrite condition (1) as follows: A node \(v \in \mathcal{N}I\) is in the answer set iff for some exit driver \(e\),

\[
S_e(k) = A_{d,e}(k) = 1 \quad \text{for some } k \geq 0. \quad \cdots \quad (2)
\]

Notice that this condition requires properties of nodes in the IP only.

Different methods can be used to utilize the above condition to extract the answers from the IP. One method which is a direct implementation is as follows: Start from the exit drivers and keep track of the successors in the IP at each iteration. The successors of an exit driver \(e\) at iteration \(i\) are in the answer set iff \(S_e(i) = 1\). This method is proposed in [B] as an extension to the method of [HN], but as mentioned there, it will require expensive set comparison operations.

Here, we will propose another method which is based on the fact that it is usually more efficient to "expect" the answers at pre-defined iteration(s) rather to "discover" them at various iterations. To illustrate the idea, let us define the factors of a selection function \(S_e(x)\) as follows:

Definition: Let \(S_e(x) = \alpha\beta^*, L_{\alpha} = [d_1, \ldots, d_k], L_{\beta} = [\beta_1, \ldots, \beta_m] \), then the factors of \(S_e(x)\) are those appearance functions \(\alpha\beta^*_i\) such that:

\[
(1) \bigwedge_{i=1}^{k} \alpha_i\beta^*_i = \alpha\beta^*; \quad i = 1, \ldots, k
\]

\[
(2) \alpha\beta^*_i \cap \alpha\beta^*_j = 0^* \text{ for } i \neq j
\]

Here, we use \(\cup\) and \(\cap\) to denote OR and AND operations on the corresponding positions respectively. Intuitively, each \(\alpha\beta^*_i\) contains exactly one occurrence of the 1's in \(\alpha\beta^*\).

Example 3: Let \(S_e = 0110(101)^*\), then the factors of \(S_e\) are:

\[
C_1^* = 0100, \\
C_2^* = 0010, \\
C_3^* = 0000(100)^*, \\
C_4^* = 0000(001)^*.
\]

We define the distance of a factor \(C^*_k = \alpha\beta^*_k\) as the number of 0's to the left of the 1 in \(\alpha\beta^*_k\). In the above example, the distances of \(C_1^*, C_2^*, C_3^*,\) and \(C_4^*\) are 1, 2, 4, and 6 respectively. We will write \(\text{dist}(C^*_k)\) to denote the distance of the factor \(C^*_k\). By definition, for \(j = 0, 1, \ldots, \)}
\[ C_i^j(n) = 1 \text{ iff } n = \text{dist}(C_i^j) + j \beta \] \quad \cdots (3)

The factors of \( S_e \) have an important meaning when processing the IP graph. Intuitively, if we include the exit driver \( e \) just before the iteration number \( i \), then we expect the answers derived from \( e \) to appear at iteration number \( i + \text{dist}(C_i^j) + j \beta \).

It is important to notice that these distances are relative to the iteration number in which \( e \) is included. So, we will include the exit driver \( e \) in the iterations as many times as the number of factors of \( S_e \), and in a way that only the nodes which will appear at specific iteration(s) are in the answer set. We will call this process "synchronization of the factors of \( S_e \)," or simply, "synchronization of \( S_e \)."

Assume that \( S_e = \alpha \beta^* \), and let \( L_a = |\alpha|, L_\beta = |\beta|, \) and \( L = L_a + L_\beta \). Moreover, let the displacement \( \Delta_i \) of the factor \( C_i^j \) be as follows:

\[ \Delta_i = L - \text{dist}(C_i^j) \]

We define the synchronized factor \( \hat{C}_i^j \) as:

\[ \hat{C}_i^j(0) = \begin{cases} C_i^j(0) - \Delta_i & \text{for } j \geq \Delta_i \\ 0 & \text{otherwise.} \end{cases} \]

The following lemma states the relationship between \( \hat{C}_i^j \) and the selection functions.

**Lemma 1:**

\[ \bigcup_{i=1}^{j} \hat{C}_i^j(n + \Delta_i) = S_e(n) \quad \text{for all } n \geq 0. \]

**proof:** It follows directly from the definitions of \( \hat{C}_i^j \) and \( S_e \).

**Example 4:** Consider the previous example, the values of \( \Delta_i \) are: \( \Delta_1 = 6, \Delta_2 = 5, \Delta_3 = 3, \Delta_4 = 1. \)

So, we can calculate \( \hat{C}_i^j \) as follows:

\[ \hat{C}_1^1 = 00000001000 \cdots \]
\[ \hat{C}_2^1 = 00000000100 \cdots \]
\[ \hat{C}_3^1 = 00000000010001 \cdots \]
\[ \hat{C}_4^1 = 0000000000100001 \cdots \]

To verify the lemma, let us consider, as an example, \( S_e(9) \) which is equal to \( \hat{C}_1^1(9 + 6) \lor \hat{C}_2^1(9 + 5) \lor \hat{C}_3^1(9 + 3) \lor \hat{C}_4^1(9 + 1) = 0 \lor 0 \lor 0 \lor 1 = 1. \)

As can be noticed from the above example, \( \hat{C}_i^j(7) = 1 \) for all \( i \). Also the synchronized factors which have periodic component have 1's only in a specific locations. In fact this is a general property, and not specific for the above example as will be proved by the next lemma.

**Lemma 2:**

\[ \bigcup_{i=1}^{j} \hat{C}_i^j(n) = 1 \text{ iff } n = L + j L_\beta; \quad j = 0, 1, \ldots \]

**proof:** See [AOx1].

Now, by combining lemma 1 and lemma 2, we can state the same property for \( S_e \) whose factors are synchronized as described before.

**Proposition 2:**

For a synchronized \( S_e = \alpha \beta^* \):

\[ S_e(n) = 1 \text{ iff } n = L + j L_\beta \quad \text{for } n \geq 0 \text{ and } j = 0, 1, \ldots \]

The above proposition implies that all the possible answers derived from an exit driver \( e \) which is "synchronized" are all the successors of \( e \) in IP that appear at iteration(s) \( L + j L_\beta \). The proof of the proposition follows directly from lemmas 1 and 2.

5. The Synchronized counting Method.

In the previous sections we presented the basic propositions on which our algorithm is based on. In this section, we will first present the algorithm in its simplest form without sophisticated optimization, then we will discuss further optimization strategies. As mentioned previously, the counting method has two phases. The first phase is the processing of the determined part DP, and the second phase is the processing of the induced part IP.

In the Synchronized counting Method (SCM), the first phase is essentially the same, except for the termination condition which is modified to handle the possible existence of cycles in the DP. The new condition is the disjunction of the old condition (counting Set \( = \phi \)) and the fact that we reach the steady state (cyclic) behavior (two sets are equal). More formally, if \( CS_i \) represents the counting set at iteration \( i \), then the first phase will terminate at iteration \( i \) when the following is true:

\[ (CS_i = \phi \text{ OR } CS_i = CS_j; \quad i > j) \]

Proposition 1 guarantees that this condition is eventually will be true for cyclic or acyclic DPs. Moreover, as we will see when discussing the complexity, \( i \leq lN_{DP}l^2 \) where \( lN_{DP}l \) is the number of nodes in DP.

The move from the first phase to the second phase is through the exit part EP. Let us define the
exit counting sets $ECS_i$ as:

$$ECS_i = CS_i \cap EP$$

For acyclic cases, the SCM will be basically the same as in the original counting method when processing the IP. However, for cyclic cases we will need some simple modifications. Assume that the DP is cyclic and that the first phase terminate at iteration number $i$. Then $CS_i = CS_0$ for some $j < i$. So, we can calculate $L_j$, $I_0$ (as defined in the previous section) as $L = i - 1$, and $I_0 = j - 1$. Also let $ECS_j$, $0 \leq i \leq L$ be the corresponding exit counting sets. It is easy to see that each occurrence of an exit driver $e$ in the ECSs corresponds to a factor of the selection function $S_e$. Let $e \in ECS_0$ (i.e., $e$ is an exit driver reachable from the $u_i$ counting set) and that $e$ corresponds to the factor $C_e^i$ of $S_e$. Then, obviously, $A_e^i = L - u$ where $A_e^i$ is the displacement of $C_e^i$ as defined before.

This means that the set of ECSs represent synchronized selection functions of all the exit drivers, and hence, proposition 2 provides a necessary and sufficient condition for termination of phase 2. This condition requires only the differentiation between the successors of the steady state (periodic) elements and those of the transient elements. In other words, the answers are all the successors that appear in iteration number $L$, and the successors of the periodic elements at subsequent iterations that satisfy $L + k (L - I_0)$, for some $k \geq 0$.

Figure 3 represents an implementation of the SCM, where in the first phase, we calculate the counting sets. If the DP is cyclic, we get also the lengths of the transient and the steady state parts, represented by $j-1$ and $i-j$ respectively. After the termination of this phase, the exit counting sets are calculated. In the second phase, the successors (in IP) of those elements in the exit counting sets that belong to the steady state part are collected in "steady_state", and the successors of those elements in exit counting sets that belong to the transient part are collected in "transient". $\psi_1$ and $\psi_2$ are temporary relations that help in collecting the successors of the steady state elements that occur after each $i-j$ iterations. Notice that if the determined part is acyclic, then the second phase is the same as the second phase of the original counting method.

The algorithm as described in Figure 3 is simple, and in fact there is a room for further optimization by considering special data structures and/or modifying the control flow slightly. For example, in phase I, the loop which compares each counting

```
begin
/* Phase I */
i = 0; CS_0 = \{s\}; phase_L_term = false; cyclic = false;
repeat
CS_{i+1} = CS_i \cap EP; i = i + 1;
if CS_i = \emptyset then begin phase_L_term = true; j = 0; end;
else begin
j = i - 1;
while j \geq 0 and not phase_L_term do
if CS_i \neq CS_j then j = j - 1
else begin phase_L_term = true;
cyclic = true; end; end; end
until phase_L_term;

/* Compute exit counting sets (ECS) */
for k = i - 1 downto 0 do
ECS_k = CS_k \cap EP;

/* Phase II */
successors = ECS_{i-1};
transient = 0;
if cyclic then begin
for k = i - 1 downto j + 1 do
successors = (successors \cap IP) \cup ECS_{k-1};
if j \neq 0 then begin for k = j downto 1 do
successors = successors \cap IP;
transient = ECS_{k-1};
for k = j - 1 downto 1 do
transient = transient \cap IP \cup ECS_{k-1};
end
steady_state = successors;
\psi_1, \psi_2 = steady_state;
phase_L_term = false;
while \psi_2 \neq \emptyset and not phase_L_term
begin for k = 1 to i - j do \psi_2 = \psi_2 \cap IP;
if \psi_2 \neq \emptyset then begin
steady_state = steady_state \cup \psi_2;
\psi_1 = \psi_1 \cup \psi_2; end
else phase_L_term = true; end;
answers = steady_state \cup transient; end;
else begin for k = i - 1 downto 1 do
successors = successors \cap IP \cup ECS_{k-1};
answers = successors; end;

Figure 3
```

set $CS_i$ with all the previous sets can be optimized by utilizing some facts from proposition 1. By this proposition, if the DP is cyclic then the period of the appearance functions will be less than or equal to $g$, which is the gcd of the lengths of the directed circuits. So, instead of comparing the current $CS_i$ with all the previous ones, we may just compare it with the previous $g$ sets. Moreover, this loop should
start only when the current CS does not add any new element since the steady state part cannot start before this happens. But evaluating $g$ itself may be relatively expensive, (in [HaN] an algorithm of $O(n)$ has been given which compute the period of a strongly connected component, where $e$ is the number of edges). Thus, we may need an estimator for the value of $g$ which can be computed easily.

A direct estimator for the value of $g$ is $\text{IN}_{D}(n)$ (the number of nodes in DP), but it may be inefficient. Another possible estimator is mentioned in [B], which is the distance between two counting sets $CS_i$ and $CS_j$, such that $CS_i \subset CS_j$. This one can be calculated very efficiently if we use binary vectors to represent the counting sets which will also make the test for equality of two counting sets very efficient.

The second phase can also be optimized with the cost of some more storage by saving the outcome of each iteration after the finishing of the transient part and check for cycles as in the first phase so that the answers can be extracted from these outcomes instead of performing the semi-join operation with the IP.

6. Complexity Analysis and Comparison

Dealing with cyclic relations in the evaluation of recursive queries efficiently is a difficult job and there are many suggestions and algorithms which are proposed in the literature. Throughout this section, $e$ and $n$ denote the number of edges and nodes, respectively, in the graph representing the input relations.

The suggestion in [HN] leads to a sound but incomplete method as proved in [B]. Although a modification has been proposed in [B], with no cost analysis, it has been mentioned there that these modifications are expensive.

[HH] proposed a method specifically for cyclic relations which takes $O(e^2)$ time. However, this method requires exponential pre-processing of the database for each update as mentioned in [HaN].

The algorithm suggested in [GSS] for cyclic cases is incomplete since the condition given for termination is not sufficient. Consider the following counter example for the same generation program and the query $sg(a, ?)$:

$$\text{up}(X, Y) = \{(a,b), (b,c), (c,d), (d,b), (a,e)\},$$

$$\text{flat}(X, Y) = \{(b,h), (e,j)\}$$

$$\text{down}(X, Y) = \{(k,i), (i,n), (j,k), (j,l), (j,n), (k,i), (j,k), (m,l), (n,m)\}.$$

For this database instance, the algorithm will produce $\{i, k, l, n\}$ as the answer. However $m$ must be also in the answer set.

[SPS] presented an algorithm which is a direct extension to the counting method and has been shown to have $O(n^3.e)$ time complexity irrespective of how the graph representing the relation is connected. In [SZ], a family of algorithms that combine the counting method and Magic Sets method [BMSU] has been suggested to suit different types of relations. So, the method has the same cost as the Magic Sets method which is $O(e^2)$.

In [HaN], an overview of a method is presented which compute only the answers that can be derived using paths which includes cycles in both the DP and IP. In the paper, the method has been claimed to have $O(n.e)$ complexity; however, it seems that the complexity is $O(n^2.e)$ since step 6 in the algorithm should be performed for each node in the set of reachable nodes by paths with cycles (i.e., $V_\epsilon$) as mentioned in the paper.

As for the algorithm presented in this paper i.e., the synchronized counting method, proposition 1 provides the upper bound for the number of iterations required until we reach the steady state (cyclic behavior). Specifically, let $N_D$, $N_I$ are the number of nodes in DP and IP, then the repeat statement in Phase I will require at most $O(N_D)$, and Phase II will require at most $O(N_D^2 + N_D.N_I)$. So the total number of semi-join operations in the worst case is $O(n^3)$.

As in [SPS], if we assume that the unit cost is the cost of retrieving a tuple from the DB, and that the semi-join operation has a cost proportional to the size of the involved counting set*, then, at most, the total cost of all the semi-join operations of all the iterations is $O(n^3)$. Also, if we assume a binary vector representation for the counting sets, then the inner loop in Phase 1 will have a cost comparable to a single semi-join operation. Adding all together, the total cost of the SCM in the worst case will be $O(n^3)$.

Another important point about SCM is that it is not required to know the type of the relations before applying it unlike most of the other extensions. The SCM has the same behavior as the original counting method in case of acyclic relations which is very efficient. However, most of the other methods proposed for cyclic cases will be expensive.

* The semi-join operation here is the same as the adj((u) operation in [SPS].
if applied to the acyclic cases.

While the worst case performance of the SCM is $O(n^3)$, as discussed above, the actual performance depends heavily on how the DP and the IP graphs are connected. In fact, the more edges (tuples in the relations) we add to these graphs, the more efficiency we get with respect to the other methods, since this helps to shorten the transient part in the appearance functions. For example, if the DP is a complete graph, Phase I will stop after exactly 3 iterations (i.e. 3 semi-join operations) irrespective to the number of nodes in the graph. This is since, starting from any node, the first iteration will give us all the nodes except the source node. In the second iteration, we get all the nodes and, in the third one the termination condition will be true. If the DP is a complete graph with self loops, then only two iterations are required. On the other hand, if the graph is cyclic with relatively few edges, then we may need $O(n^2)$ iterations. An example of such a graph is given in Figure 4.

![Graph](image)

Figure 4

For this graph, if we start with $CS_0 = \{a\}$, then $CS_1 = \{b\}$, ..., $CS_4 = \{e\}$, $CS_5 = \{a, b\}$, ..., $CS_9 = \{a, b, e\}$, ..., $CS_{13} = \{a, b, d, e\}$, ... . So, the size of the counting sets will increased by one after each fourth (i.e. $n - 1$) iteration.

7. Conclusion

We presented a method for evaluating linear recursive queries, called the "Synchronized Counting Method". SCM is a direct extension to the counting method [BMSU], but is applicable to cyclic queries as well as acyclic one. The method is proved to be sound and complete in the general case. Worst case analysis shows that $O(n^3)$ semi-join operations may be needed in the cyclic case. The synchronized counting method compares favorably with the other methods available in the literature for evaluating linear recursive queries.

References


373