PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

ANSWERING LINEAR RECURSIVE QUERIES IN CYCLIC DATABASES

Ching-Shyan Wu

Manager
Div. of Application Developing
Information Service Center
Veterans General Hospital, VACRS
Shih-pai, Taipei, Taiwan, R.O.C.

Lawrence J. Henschen

Professor
Dept. of Electrical Engineering
and Computer Science
The Technological Institute
Northwestern University

ABSTRACT

In this paper we propose a method to uniformly handle cyclic and acyclic data relations in the linear recursive queries. The idea to compute the level informations called the **recurrence sequences** (RSs) is based on our developed theorems for managing cycles. The computations are limited in a connected component of a relation. The final representation of RSs is the union of finite number of sets, each of which is a set driven by a value, namely the **virtual cycle**. Then, the answer determination is reduced to check the emptiness of intersection of pairs of RSs by choosing a bridge point in the left-hand side relation and another vertex (reachable from the bridge point) in the right-hand side relation. In virtue of such kind of answer determination, we call it the *intersection* approach to distinguish from the *enumeration* approach which will be published later. Shifting some computation to the compiling phase, we can improve the performance to O(ne), where e is the total number of accessed edges and n is the total number of vertices in the corresponding connected graphs.

1 INTRODUCTION

Queries on predicates defined by linear recursive rules in an IDB is a field in the deductive databases (Gallaire et al. 1984). Several methods have been proposed to deal with such problem, however, processing in an efficient manner is still a challenge for researchers (Henschen and Naqvi 1984, Bancilhon and Rammakrishnan 1986, Han and Henschen 1986a, 1986b and 1987). An elegant method like the counting method may perform efficiently on those EDB relations in an asynchronous structure, but may not perform well in a rather complicated structure such as a cyclic data base or a combination of asynchronous and cyclic data structures. The main shortcoming of the counting method is that in order to compute the all answers many intermediate tuples, that although directly related to the query constant are not answers to the query, are generated as a by-product during the processing of a query. This not only increases the space of the intermediate relation but also causes delay for the user. Here is an example that depicts such a problem. Suppose the data related to the query R(a, ?) in the extensional relations are shown as the content of Table 1 and the virtual relation R is defined by the following rules:

R(x, y) :- A(x, y)....................................(1)
R(x, y) :- B(x, u), R(u, v), C(v, y).............(2)

B	(a,b) (b,b1) (b1,b2) (b2,b3) (b3,d) (d,b4) (b4,b5) (b5,b) (b,b6) (b6,b7) (b7,b) (a,c) (c,c1) (c1,e) (e,d) (e,g) (g,c2) (c2,c) (c,f) (f,g) (f,c)
A	(d,h)
C	(h,h1) (h1,h2) (h2,k) (h2,k1) (k,h3) (h3,h) (k,k2) (k1,k2) (k2,k3) (k3,k4) (k4,k5) (k5,k) (k,k2)

Table 1 Related data of the Query Constant a.

It is well-known that the answers are the union of values derived by the formulas B^iAC^i. An important concept is that of level; for example a value b such that $B^i(a,b)$ has level i in relation B *w.r.t.* the query constant a. In the counting method, to involve level information the rules are rewritten as follow:

i) count(a,0)
ii) count(x,i) :- count(y,j),B(y,x),i=j+1
iii) R'(x,y,i) :- count(x,i),A(x,y)
iv) R'(x,y,i) :- count(x,i),B(x,u),R'(u,v,j),C(v,y),i=j-1

The data related to a can be drawn as a graph, called the **query-graph**, as shown in Figure 1.

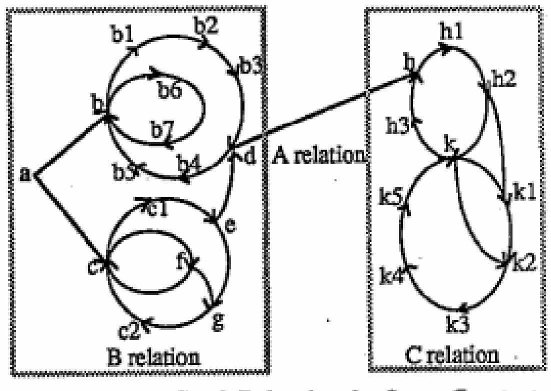

Figure 1 Query Graph Related to the Query Constant a.

To answer the query, the counting method has to compute the following data tuples in the count relation. Notice that values answering in cycles occur at infinitely many levels *w.r.t.* a; for example b occurs at level 1, 4, 7, 8, ..., *etc.*

(a,0)	(b,1)	(b1,2)	(b2,3)	(b3,4)	(d,5)	(b4,6)	(b5,7)	(b,8)	...
		(b6,2)	(b7,3)	(d,4)	(b4,5)	(b5,6)	(b,7)	(b1,8)	...
				(b,4)				(b6,8)	...
		...			...			...	
	(c,1)	(c1,2)	(c,3)	(d,4)	(c,5)	(c1,6)	(c,7)	(d,8)	...
						(f, 6)	(c,7)	...	
		(f,2)	(g,3)	(c2,4)	...				

Table 2 Some Tuples of Relation *count*.

These tuples are just a minor portion of all the tuples generated in the relation count. The first tuple generated for the modified relation R' is (d,h,4), from which the first two deduced answers are h3 and k2, which are of level 4. However, the answers to this case are all the data values appearing in the right-hand side of Figure 1. Therefore, one might imagine that a huge number of tuples must be put into the count relation as well as the modified relation R'. This case illustrates how ineffective the counting method is on treating the data in a complicated structure. To overcome such a problem, some authors proposed two algorithms to handle the different structures: one for the acyclic and the other for the cyclic, for examples, the magic set method and counting methods (Bancilhon et al. 1986) and the LM and LCM methods (Han and Henschen 1987). Such kind of processing needs users to decide which one to apply. Besides, they also suffer efficiency problem. The method proposed by Haddad and Naughton (1988) improves efficiency but makes some restriction on the structures of the two base relations. Because they involve some extra levels into the so-called distance sets, compared to the recurrence sequences introduced in this paper, the two relations B and C must be cyclic. Our approach is an algorithmic computation that directly explores the relevant data values in the base relations starting from the query constant. The cycle information is first obtained by a graph traversal algorithm (Johnson 1975, Mateti and Deo 1976, Tarjan 1973). The cycle tree associated with some vertex is then constructed to express the relationship between the data values and the corresponding cycle information and is used to generate a set of integer values, called the **recurrence sequence**, related to a vertex in a graph. The recurrence sequence represents all possible occurrences of levels of the vertex *w.r.t.* the query constant (or origin). Finally, the answers to the query can be determined by using the recurrence sequences of bridge vertices (see section 3). This is somewhat different from previous well-known methods. Our method can be implemented as a subprogram. When the compiler has detected a defined predicate to be linearly recursive, no compilation output forms are needed. Instead a calling statement is attached to that predicate. At the time of a query, the inquired predicate and the query constant will be passed as parameters to this subprogram for evaluating the answers. Moreover, our method can be implemented to execute in parallel.

We will organize sections of this paper in the following manner. In section 2, we give a sequence of lemmas and theorems to figure out the idea how to manage the cycle information into a simple form. The scheme to compute the level information will be described in section 3. The algorithm and complexity analysis is given in section 4. In section 5, we make a brief concluding remarks. In the following description, unless otherwise specified we follow the notational conventions. Upper case letters A, B, and C denote the extensional relations; the lower case letter a is the query constant; and the lower case letters x, y, z, u, v, and w, represent variables. We assume that the query is of the form R(a, ?) in the canonical type as defined by above equations 1 and 2 (Han and Henschen 1986a).

2 MANAGING CYCLES

The levels of a data value (or vertex) driven by a single cycle can be enumerated by adding multiples of the length of the cycle. However, for a vertex driven by several cycles, it becomes harder to list them without duplication. For example, if there are n cycles of lengths $c_1, c_2, \ldots, c_n$, respectively, then all of the possible levels can be denoted as:

$$\{k_1 \cdot c_1 + k_2 \cdot c_2 + \ldots + k_n \cdot c_n \mid k_i \geq 0, i = 1, \ldots, n\}.$$

It appears that there may be different ways to generate some given level. The larger the value of a level, the more combinations can be expressed. Thus, it is important to avoid duplicate enumeration. Let us first observe some examples.

1. $\{k_1 * 5 + k_2 * 3\} = \{0, 3, 5, 6, 8, 9, 10, 11, \ldots\}$
   $= \{0, 3, 5, 6\} \cup \{8 + k * 1\}$,
2. $\{k_1 * 6 + k_2 * 8\} = \{0, 6, 8, 12, 14, 16, \ldots\}$
   $= \{0, 6, 8\} \cup \{12 + k * 2\}$,
3. $\{k_1 * 5 + k_2 * 3 + k_3 * 7\} = \{0, 3, 5, 6, 7, \ldots\}$
   $= \{0, 3\} \cup \{5 + k * 1\}$,
4. $\{k_1 * 4 + k_2 * 5 + k_3 * 6 + k_4 * 26\}$
   $= \{0, 4, 5, 6, 8, 9, 10, 11, \ldots\}$
   $= \{0, 4, 5, 6\} \cup \{8 + k * 1\}$.

From these examples, we find that there is a common situation. A regular pattern (in an arithmetic sequence) always appears after some initial irregular values no matter what the number of cycles. They have been reduced almost to a single cycle enumeration. This motivates us to approach the general case with n cycles. Before discussing, we need some terminology and definitions. We use the following notational convention only for this entire section. Lower case letters stand for nonnegative integers and upper case letters for sets of nonnegative integers. The multiplication operator is omitted in an arithmetic expression.

Definition: $F + G = \{f + g \mid f \in F, g \in G\}$.

Definition: If $C_1, \ldots, C_n$ are cycles with lengths $c_1, \ldots, c_n$, respectively. Then Define $F(c_1, c_2, \ldots, c_n)$ to be the set $\{f + k_1c_1 + \ldots + k_nc_n \mid f \in F, k_i = 0, 1, \ldots, i = 1, \ldots, n\}$. For the special case, $F = \{f\}$, we write $f(c_1, c_2, \ldots, c_n)$ in place of $\{f\}(c_1, c_2, \ldots, c_n)$ and call f the **start point** (or value) of the n cycles.

Definition: $a * F = \{af \mid f \in F\}$. '$a * F$' is denoted as '$a F$' if there is no ambiguity.

Definition: $F * a = \{fa \mid f \in F\}$.

2.1 Properties

We give the following properties without proofs. (They can be proved easily by set theory.)

(1) $F + G = G + F$.
(2) $a * F = F * a$.
(3) $(F + G) + H = F + (G + H)$.
(4) $(ab) * F = a * (b * F)$.
(5) $a * (F + G) = (a * F) + (a * G)$.
(6) $F(c) = \cup \{f(c) \mid f \in F\}$.
(7) $(F \cup G) + H = (F + H) \cup (G + H)$.
(8) $a * (F \cup G) = (a * F) \cup (a * G)$.
(9) $F(c) = F + 0(c)$.
(10) $F(c_1, \ldots, c_n) = F + 0(c_1, \ldots, c_n)$.
(11) $F(c_1, \ldots, c_n) = F(c_1, \ldots, c_{n-1}) + 0(c_n)$.
(12) $a * [F(c)] = [a * F](ac)$.
(13) $a * [F(c_1, \ldots, c_n)] = a * F(ac_1, \ldots, ac_n)$.

2.2 Cycle Merging Theorem

In number theory (Niven and Zuckerman 1966), it is well-known that for any integers a and b, with $b > 0$, there uniquely exists integers q and r such that $a = qb + r$, $0 \le r < b$. If d is the greatest common divisor (gcd) of a and b, then there exist infinitely many pairs (x,y) such that $ax + by = d$. If both a and b are nonnegative, then exactly one of x and y must be negative. We prefer to those with the value of x negative, and in the following presentation we denote the pair (x,y) to be (-u,v), both u and v are nonnegative integers. We state the lemmas and theorems needed to develop our method, but refer the reader to (Wu 1988) for most of the proofs.

Lemma 1. *Let $p, q > 1$. If $gcd(p,q) = 1$ then there exists a nonnegative pair (u_0,v_0) such that u_0 less than q and v_0 greater than 0 and $-u_0 p + v_0 q = 1$.*

From now on, assume that p is relatively prime to q, *i.e.* $gcd(p,q) = 1$.

Lemma 2. *For any $t \ge 0$ there exist $k_1, k_2 \ge 0$ such that $(p-1)(q-1) + t = k_1 p + k_2 q$.*

Lemma 3. *There are no $k_1, k_2 \ge 0$ such that $(p-1)(q-1)-1 = k_1 p + k_2 q$.*

Lemma 2 shows that any integer greater than or equal to $(p - 1)(q - 1)$ can be generated by the two values p and q, and lemma 3 guarantees the value $(p - 1)(q - 1) - 1 = p q - p - q$ not in $0(p, q)$. Thus combining the result of the two lemmas, we see that $(p - 1)(q - 1)$ is the beginning value with cycle length of 1 and there are only finite values in $0(p, q)$ less than $(p - 1)(q - 1)$. For any two integers $m, n > 0$, the property also holds. Thus, we have the following two theorems.

Theorem 1. *$0(p,q) = F \cup e(1)$, where $e = (p - 1)(q - 1)$ and F is a finite set of integers less than (e-1). Moreover, F and e are uniquely determined.*

Theorem 2. *Let $m, n > 0$ and $gcd(m, n) = r$. Then, for any $t \ge 0$, $r[(p - 1)(q - 1) + t] = k_1 m + k_2 n$, where $p = m/r$ and $q = n/r$, for some integers $k_1, k_2 \ge 0$. That is, $0(m,n) = F' \cup e'(r) = rF \cup re(r)$, where $F' = rF$ and $e' = re$, and F and e are defined as in the theorem 1, i.e. $e' = lcm(m,n) + gcd(m,n) - m - n$, where lcm and gcd stand for the least common multiplier and the greatest common divisor, respectively.*

Lemma 4. *If $gcd(p_1, p_2, \ldots, p_n) = 1$ then there exists a unique finite set F and an integer d such that*
(1) $0(p_1, p_2, \ldots, p_n) = F \cup d(1)$.
(2) $\forall f \in F$, $f < d - 1$

Theorem 3. *(Cycle Merging Theorem) Let $b \ge 0$ & $c_i > 0$, $i = 1, \ldots, n$. Then there exist a finite set F and an integer d such that $d \ge 0$ and*
(1) $b(c_1, \ldots, c_n) = F \cup d(r)$ and
(2) $\forall f \in F, f < d - r$, where $r = gcd(c_1, \ldots, c_n)$.
And, furthermore, F and d are uniquely determined.

Proof. Without loss of generality, assume all c's are distinct values.
By properties 10, 11, and 12,
$b(c_1, \ldots, c_n) = b + 0(c_1, \ldots, c_n)$
$= b + [0(p_1, \ldots, p_n) r]$,
where $c_i = p_i r$, $i = 1, \ldots, n$. Then
$gcd(p_1, \ldots, p_n) = 1$.
By lemma 5, there are unique F' and d' satisfying the conditions. So,
$b(c_1, \ldots, c_n) = b + [[F' \cup d'(1)] r]$
$= b + [rF' \cup rd'(r)]$, by properties 2, 5 and 12,
$= [b+rF'] \cup \{b+rd'\}(r)$, by properties 1 and 7.
Let $F = b + rF'$ and $d = b + rd'$. Then F and d are also uniquely determined.
This proves the theorem. Q.E.D.

We will call the final merged cycle with length r the **virtual cycle**, the smallest value d the **start recurring point** (srp) of the virtual cycle, and call the first finite values less than the srp in $b(c_1, c_2, \ldots, c_n)$ the **F-set**.

2.3 Formula for Merging n Cycles

In theorem 2, the srp of $0(m,n)$ can be computed by the expression $r(p-1)(q-1)$ which is really the value $lcm(m,n) + gcd(m,n) - m - n$, where lcm is the abridgement of the least common multiplier. We want to extend this formula to compute general cases. Suppose we want to compute the srp of $0(18, 12, 21, 27)$, from properties of the defined operators, we may first compute the srp of $0(18, 12)$, which can be expressed as $\{0\} \cup 12(6)$, then compute $12(6, 21)$ to be $\{12, 18, 24\} \cup 30(3)$ and finally compute $30(3,27)$, which is the same as $30(3)$. Since 27, 24, 21 and 18 are in $0(18, 12, 21, 27)$, the srp of this example must be 18. Therefore, $0(18, 12, 21, 27) = \{0, 12\} \cup 18(3)$. Below are the step-by-step computations.

$$\begin{aligned}
&0(18, 12, 21, 27) = 0(18, 12) + 0(21) + 0(27)\\
&= [\{0\} \cup 12(6)] + 0(21) + 0(27)\\
&= [0(21) \cup 12(6, 21)] + 0(27)\\
&= [0(21) \cup \{12, 18, 24\} \cup 30(3)] + 0(27)\\
&= 0(21, 27) \cup \{12, 18, 24\}(27) \cup 30(3, 27)\\
&= 0(21, 27) \cup \{12, 18, 24\}(27) \cup 30(3)\\
&= \{0, 12, 18, 21, 24, 27\} \cup 30(3)\\
&= \{0, 12\} \cup 18(3).
\end{aligned}$$

Note that in each step of the above computations, we focus only on merging the virtual cycle with next cycle. Thus, to compute the srp, we may take advantage from temporarily ignoring the intermediate F-sets of the previous steps. This idea is generalized for the derivation of n cycles as shown below.

$$
\begin{aligned}
&O(c_1, c_2, \ldots, c_n) = O(c_1, c_2, \ldots, c_{n-1}) + O(c_n)\\
&= O(c_1, c_2, \ldots, c_{n-2}) + O(c_{n-1}) + O(c_n)\\
&= \quad \cdots\\
&= O(c_1, c_2) + O(c_3) + \ldots + O(c_{n-1}) + O(c_n)\\
&= [F_1 \cup d_1(g_1)] + O(c_3) + \ldots + O(c_{n-1}) + O(c_n)\\
&= [F_1(c_3) \cup d_1(g_1, c_3)] + O(c_4) + \ldots + O(c_{n-1}) + O(c_n)\\
&= [F_1(c_3) \cup F_2 \cup d_2(g_2)] + O(c_4) + \ldots + O(c_{n-1}) + O(c_n)\\
&= [F_1(c_3, c_4) \cup F_2(c_4) \cup d_2(g_2, c_4))] + O(c_5) + \ldots +\\
&\quad O(c_{n-1}) + O(c_n)\\
&= \cdots\\
&= [F_1(c_3, \ldots, c_{n-1}) \cup F_2(c_4, \ldots, c_{n-1}) \cup F_3(c_5, \ldots,\\
&\quad c_{n-1}) \cup \ldots \cup F_{n-3}(c_n) \cup F_{n-2} \cup d_{n-2}(g_{n-2})] + O(c_n)\\
&= F_1(c_3, \ldots, c_n) \cup F_2(c_4, \ldots, c_n) \cup F_3(c_5, \ldots, c_n) \cup\\
&\quad \ldots \cup F_{n-2}(c_n) \cup d_{n-2}(g_{n-2}, c_n)\\
&= F_1(c_3, \ldots, c_n) \cup F_2(c_4, \ldots, c_n) \cup F_3(c_5, \ldots, c_n) \cup\\
&\quad \ldots \cup F_{n-2}(c_n) \cup F_{n-1} \cup d_{n-1}(g_{n-1}).
\end{aligned}
$$

The last expression can be simplified to $F \cup d(g_{n-1})$, for some value $d \le d_{n-1}$. The g's and d's can be expressed as following:

$$
\begin{aligned}
g_1 &= \gcd(c_1, c_2)\\
d_1 &= \mathrm{lcm}(c_1, c_2) + g_1 - c_1 - c_2\\
g_2 &= \gcd(g_1, c_3)\\
d_2 &= d_1 + \mathrm{lcm}(g_1, c_3) + g_2 - g_1 - c_3\\
&= \mathrm{lcm}(c_1, c_2) + \mathrm{lcm}(g_1, c_3) + g_2 - c_1 - c_2 - c_3\\
&\quad \cdots\\
g_i &= \gcd(g_{i-1}, c_{i+1})\\
d_i &= d_{i-1} + \mathrm{lcm}(g_{i-1}, c_{i+1}) + g_i - g_{i-1} - c_{i+1}\\
&= \mathrm{lcm}(c_1, c_2) + \mathrm{lcm}(g_1, c_3) + \mathrm{lcm}(g_2, c_4) + \ldots +\\
&\quad \mathrm{lcm}(g_{i-1}, c_{i+1}) + g_i - c_1 - \ldots - c_{i+1}\\
&\quad \cdots\\
g_{n-1} &= \gcd(g_{n-2}, c_n)\\
d_{n-1} &= d_{n-2} + \mathrm{lcm}(g_{n-2}, c_n) + g_{n-1} - g_{n-2} - c_n\\
&= \mathrm{lcm}(c_1, c_2) + \mathrm{lcm}(g_1, c_3) + \mathrm{lcm}(g_2, c_4) + \ldots +\\
&\quad \mathrm{lcm}(g_{n-2}, c_n) + g_{n-1} - c_1 - \ldots - c_n
\end{aligned}
$$

From these formulas, we conclude that for some i, $d_i = d_{i+1}$ if and only if $g_i = g_{i+1}$, *i.e.* c_{i+2} has no contribution to reduce the virtual cycle during the computation. If neither can it span any value to decrease the d_{n-1} value then it can be eliminated from the n cycles. Such a cycle is *useless*. Thus, for any set of n cycles, eliminating all those useless cycles from the n cycles can be handled by this manner. To further reduce computing, we found some result (Wu 1988) that can help eliminating some computation to get the srp value faster. In the lemma 3, we already have that pq - p - q is not in O(p,q). But, if this value can be generated by latter cycles in the cycle set (p, q, ...), we discover that the values from pq - p - 2q + 1 to pq - p - q are all in O(p,q), provided that p > q, and a similar result for O(m,n) also follows in turn. The readers who want to know more detail of this result are encouraged to read the reference. The problem to decide the emptiness of the intersection of any two cycles b and d with their start points a and c, respectively, is pretty easy.

Theorem 4. *(Cycle Intersection Theorem) Given a, c ≥ 0 and b, d > 0. Then, 1) $a(b) \cap c(d) \neq \phi$ if and only if $|a(b) \cap c(d)| = \infty$ and 2) $a(b) \cap c(d) \neq \phi$ if and only if gcd(b,d) is a divisor of the absolute difference of a and c.*

3 ALGORITHMIC COMPUTATION

For the sake of easily describing the algorithms, we need some additional terminology and definitions. We also restrict a graph to be a connected graph, because only those nodes related to the query constant that forms a connected component, as shown in Figure 1, are needed to answer a query. A graph is denoted as G=(V, E), where V is the set of vertices (or nodes) each of which is a value appearing in some edges and E is the set of edges that correspond to tuples in a relation.

3.1 Definitions

Definition: A path from u to v is a sequence of vertices denoted as $(v_0, v_1, \ldots, v_{n-1}, v_n)$ s.t. 1) $v_0 = u$ and $v_n = v$ and 2) $(v_i, v_{i+1}) \in E$, i = 0, 1, ..., n-1. The length of a path from u to v will be called a level of v *w.r.t.* u. A **simple path** is a path that does not revisit any vertex in the path. A path that has $v_1 = v_n$ and does not re-visit any other vertex in the path is called a **simple cycle**. Note that in the remainder of this paper all references to cycles are to be understood as references to simple cycles.

Definition: In a graph, the start vertex with respect to which the levels of other vertices (including itself) are defined is called the **origin** of the connected graph. Thus, origins are the only vertices that have level 0 and can be used to identify connected graphs in a relation.

Example of the origin is the query constant; and so is each of the bridge vertices in the C relation Figure 1 has the origins a and h. Note again that in the processing to be described below, we split the query-graph into several connected graphs: one in relation B and some in relation C. Thus, in Figure 1 a is the origin in the B relation and h is the origin in the C relation. We may also refer to the a-graph and the h-graph, respectively.

Definition: For any vertex v, the set of all possible levels of v *w.r.t.* itself, *i.e.* the levels from v to v, is called the **self- recurrence sequence** of v, denoted as SRS(v). Actually, SRS(v) contains either a singleton or infinite many elements. When SRS(v) is a infinite set, v must be in some cycles.

Definition: The **recurrence sequence** of v with respect to a vertex p is defined as the set of all possible levels from p to v, denoted as $RS_p(v)$. We use RS(v) to stand for the recurrence sequence of v *w.r.t.* the origin of the graph containing v, if no ambiguity arises.

Definition: Define relation ©: V → V as $(v_1, v_2) \in$ © iff either $v_1 = v_2$ or there is a cycle (not necessarily simple cycle) that passes through v_1 and v_2.

Clearly, © is an equivalence relation on V, and each equivalence class is a strongly connected component (SCC). In the left-hand side of Figure 1 has three SCCs, namely, {a}, {b,b1,b2,b3,d,b4,b5,b6,b7} and {c,c1,e,g, c2,f}, while the entire of right-hand side is an SCC.

Definition: A vertex v is called a **pivot** iff v is contained in at least one simple cycle and either v is the query constant or there is an edge (v',v) such that v' and v are in different SCCs.

Definition: A vertex v in an SCC is said to be a **joint** iff it is contained in more than one simple cycle. A vertex may be a pivot as well as a joint at the same time. For example, vertices b and c in Figure 1 are the cases.

The recurrence sequence is an important appliance in deciding the answers to a query. This can be seen that a vertex v in relation C is an answer if and only if there is a bridge (p,q) in A satisfying that the intersection of $RS_a(p)$ and $RS_q(v)$ is not empty. The following procedure describes a general procedure to compute the RSs.

```
PROCEDURE (RSEVAL): For a given v in V, to evaluate RS(v).
    1. For each simple path from the origin to v do
        1.1 for each u in the path, compute SRS(u),
        1.2 merge the path length with each of the SRSs
            computed in 1.1, call it PRS(v),
    2. Union the set of the PRSs to obtain RS(v).
```

Such a way of computation is, of course, time-consuming, for there is a lot of duplication. This can be reduced by having step 1.1 compute SRSs only on those pivots and joints in the paths. Before describing the algorithm to compute these SRSs from the cycles that exist in the class (or SCC), we need to introduce the concept of a cycle tree.

3.2 SRS Computation and Cycle Trees

The SRS computation is somewhat relevant to the structure of cycles in an SCC. To analyze the cycle structure, we transform the original graph to a new one by regarding each cycle in the graph as a vertex. An edge links two vertices in the transformed graph if the original represented two cycles have some vertices in common. According to this transformation, then, there are two kinds of structure in graphs. We will show examples in section 3.3 and compute their SRSs.

Definition: A **cycle tree** associated with a vertex v in a graph is a tree satisfying 1) v is the root of the tree, 2) each of the second level nodes of the tree is a cycle containing v, 3) each of the other nodes is also a cycle attaching to the cycle of its parent node, 4) in any path a cycle appears at most once, and 5) each link (C_1,C_2) in the cycle tree is labeled by a value to stand for the length of the cycle C_2.

```
PROCEDURE (CTCONST): To construct the cycle tree of vertex v in an SCC.
    1 Generate the root node for v,
    2 For each cycle containing v do
        2.1 set up a node for the cycle, say N
        2.2 pick up all the cycles attaching to N except
            those already appearing in the current path
            leading from the root down to N,
        2.3 if this is not empty then for each cycle picked
            up, do the same as the step 2.1 and 2.2.
```

This systematical construction of the cycle tree, although, excluding a cycle appearing twice in a single path of the cycle tree does have some cycles appearing in several paths of the cycle tree. Since computing the SRS for a vertex is the intention, it seems not necessary to count on every cycle because the various ways to generate a level value are not important, what the matters is whether or not a level can be generated at all. Thus, it is important to have heuristics to cut off a path during construction of a cycle tree. The purpose is to keep it small but enough to generate the desired SRS. We will introduce an example to explicate our strategy.

3.3 Examples of SRS Computation

1) Simple Structure of Cycles

This kind of structure is characterized by that the transformed graph contains no loops, *i.e.* it is a DAG or tree. The structure in the left-hand side of Figure 1 is such an example. Assume the cycle information has been found and named to be 1 to 5. Cycle 1 is b-b6-b7, cycle 2 b-b1-b2-b3-d-b4-b5, cycle 3 c-f, cycle 4 c-f-g-c2, and cycle 5 c-c1-e-g-c2. Then, the three pivots, b, c, and d, have their cycle trees shown in Figure 2, respectively.

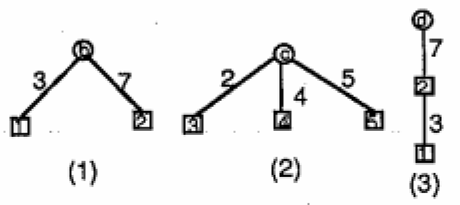

Figure 2 Simple Structure of Cycles.

The SRSs of b and c can be computed just by simple merging of the related cycles, for the cycles appear on the same level of the cycle tree:

$$SRS(b) = 0(3, 7) = \{0, 3, 6, 7, 9, 10\} \cup 12(1),$$
$$SRS(c) = 0(2, 4, 5) = \{0, 2\} \cup 4(1).$$

The SRS of d needs a little effort to compute, because the two cycles are not on the same level. In order to include cycle 1, it is necessary to go around cycle 2 at least once. So the computation looks like:

$$SRS(d) = 0(7) \cup 7(7, 3) = \{0\} \cup 7(7, 3)$$
$$= \{0, 7, 10, 13, 14, 16, 17\} \cup 19(1).$$

The start value 7 in the second set 7(7,3) is used to express necessary one traversal of cycle 2. The b-tree and c-tree are flat structures and d-tree is a path structure.

2) Loop Structure of Cycles

This is interesting and complicated because the transformed graph also has some cycles. Let us observe Figure 3. This graph contains 9 cycles. The first 7 cycles are connected together to form a ring. The numbers in the center and northwest corner of each cycle denote, respectively, the cycle number and its length. Two additional cycles are formed by the inner and outer paths. Assume the inner and outer cycles are cycle 8 and 9 with their lengths 16 and 26, respectively.

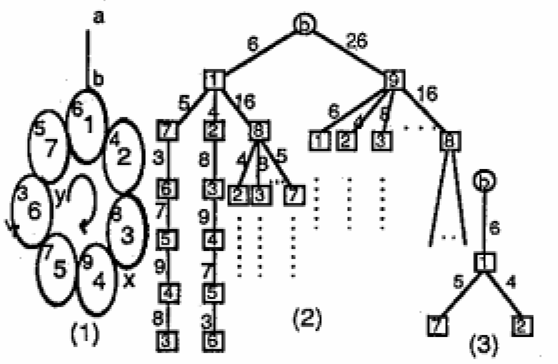

Figure 3 Ring Structure of Cycles.

Part of the cycle tree of vertex b is shown in Figure 3(2), whose size exceeds the need for computing the SRS of vertex b. The strategy to reduce the size of cycle trees is devised as follow. Let a vertex v be a pivot/joint in some SCC H. Then, the clue is the gcd of the lengths of all the simple cycles in H, which will be named as the virtual cycle length (vcl) of H and denoted as vcl(H). After computing vcl(H), proceed below steps. First, at current constructed node pick up all its child cycles and then compute the current gcd of the lengths of the picked-up cycles and the srp which includes the so-far path length. In the mean time, mark the useless cycles from the set of picked-up cycles. Second, check if the current gcd is equal to vcl(H). If not, then sort the recently picked cycles in an ascending sequence of their lengths and visit the one with the smallest length and apply the above procedure. Third, if already reach the vcl(H), then include only the lower cycles in the tree that can contribute some levels to the F-set which is less than the current srp. Those cycles without contribution are terminated the following branches. Last, try to involve the other unexplored nodes (in the picked-up set of cycles.) If any of them has some contribution, construct it a node and expand the following paths otherwise terminate the path. For those marked useless can be further deleted from the tree. This means that the ultimate constructed tree contains only those cycles really contributing some levels to the SRS(v). We show the process step by step for SRS(b) in the second example.

1. The virtual cycle is first computed by gcd(6, 4, 8, 9, 7, 3, 5, 16, 26) = 1.
2. At node b, pick cycles 1 and 9: 0(6,26) = {0, 6, ..., 44} ∪ 48(2). The current virtual cycle is of length 2 with srp equal to 48 and no cycle is useless. Then, select the smaller cycle 1 (node 1) to expand.
3. At node 1, pick cycles 2, 7, and 8: 6(4, 5, 6, 16, 26) = {4, 5, ..., 12} ∪ 14(1). The current virtual cycle is found at this step and cycles 8 and 9 are marked as useless, and 14 is so far the current srp. Therefore,
4. at node 2, pick cycle 3 and discard it because traversing this cycle even once gives a length greater than 14 and terminate this path expansion,
5. at node 7, we discard cycle 6 for the same reason and terminate this path,
6. at node 8, we terminate this path and can even delete the node because the cycle 8 is useless. Then, return to node 1,
7. at node 9, discard it (because useless) and terminate the path.

The final cycle tree is shown in Figure 3(3) and
SRS(b) = {0, 6, 10, 11, 12} ∪ 14(1).

3.4 Example of RS computation

To compute the RS, we continue to use the second example to show how it can be done. Assume there is a vertex v in cycle 6 which is a bridge point for the query, so that its recurrence sequence is needed. Let x and y be the lengths of outer and inner simple paths from a to v, respectively (these paths are shown in bold in Figure 3.) Let v1, v2, ..., v7 be the joints of c1 and c2, c2 and c3, ..., c7 and c1, respectively. The SRS of these 7 joints are computed as below.

SRS(v1) = 14(1) ∪ {0, 4, 6, 8, 10, 11, 12}
SRS(v2) = 14(1) ∪ {0, 4, 8, 10, 12}
SRS(v3) = 16(1) ∪ {0, 8, 9, 12}
SRS(v4) = 13(1) ∪ {0, 7, 9, 10}
SRS(v5) = 6(1) ∪ {0, 3}
SRS(v6) = 8(1) ∪ {0,3, 5, 6}
SRS(v7) = 10(1) ∪ {0, 5, 6, 8}
SRS(v) = 6(1) ∪ {0, 3}

Then,
RS(v) = {x + SRS(b) + SRS(v1) + SRS(v2) + SRS(v3) + SRS(v4) + SRS(v5)} ∪ {y + SRS(b) + SRS(v1) + SRS(v7) + SRS(v6) + SRS(v5)}

This reduces to
RS(v) = x + (6(1) ∪ {0,3,4}) ∪ y + (6(1) ∪ {0,3,5})
= min{x,y}+ 6(1) ∪ {x,y,x+3,y+3,x+4,y+5}.

The PRSs are obtained by merging the SRSs, and RS is then computed by unioning the PRSs. The union operation is rather simpler to manage than the merging operation. Here, two methods are proposed to merge the SRSs. From the Cycle Merging Theorem, the final result of SRS(u) and SRS(v) can be expressed by F_1 ∪ a(b) and F_2 ∪ c(d), respectively, where F_1 and F_2 are two finite sets and a, b, c, and d are positive integers. Then,

SRS(u) + SRS(v)
= {F_1 ∪ a(b)} + {F_2 ∪ c(d)}
= {[F_1 ∪ a(b)] + F_2} ∪ {[F_1 ∪ a(b)] + c(d)}
= (F_1 + F_2) ∪ a + F_2(b) ∪ c + F_1(d) ∪ {a+c}(b,d)
= F ∪ e(f),

where f = gcd(b,d), e = (a+c)+lcm(b,d)+gcd(b,d)-b-d, and F is the set containing the values less than e and is obtained from (F_1+F_2) ∪ a+F_2(b) ∪ c+F_1(d) ∪ {a+c}(b,d). The program to perform this calculation is simple but the space for the F-set may become huge if there is a large number of SRSs to merge. Another way is directly adopting the cycle trees. Assume that the final cycle trees of SRS(u) and SRS(v) are CT_1 and CT_2, respectively. Then, the merging is done in the following manner:

PROCEDURE (CTMERGE): to merge two cycle trees CT_1 and CT_2.

1. Let f be the gcd of all the cycles in CT_1 and CT_2.
2. Construct a virtual node x as a root.
3. Link each of subtrees of CT_1 and CT_2 to x.
4. Traverse and reduce the merged tree in the same manner as noted above.

The second approach can be easily extended to deal with the merge of any finite number of SRSs, because the final representation is also a tree. Thus, it is generally better than the first one. The main idea is to pick all of the cycles in the first level of each cycle tree as the first level cycles of the virtual node, then, traverse down the second level in the same manner as before to construct a cycle tree.

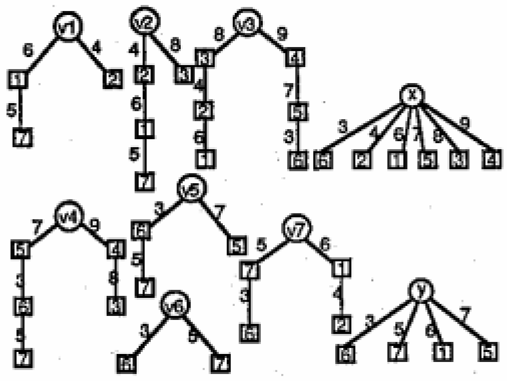

Figure 4 Example of Cycle Tree Merging.

Considering the previous example again, the result of cycle trees merging on the two paths, x and y is shown in Figure 4. The trees headed by v1, v2, ..., and v7 in the left-hand side of Figure 4 are the cycle trees, respectively, to denote their SRSs. The cycle tree for the node b is depicted in the Figure 3(3). Recall that nodes b, v1, v2, v3, v4 and v5 are merged for the x path and b, v1, v7, v6 and v5 for the y path. The two merged trees, rooted by x and y, respectively, are the PRSs of paths x and y. The x-tree can be further reduced to contain only cycles 6 and 2, and the y-tree to contain cycles 6, 7 and 5. This is because in the x-tree cycles 1, 5, 3 and 4 can be detected useless, and the same reason for the y-tree.

From the above discussion, we summarize that if any path from the origin to a vertex passes by some SCCs then the path can be associated with a virtual cycle with the length equal to the gcd of virtual cycle lengths of the visited SCCs. Thus, we have the following two lemmas.

Lemma 5 *For each vertex v in an a-graph, a path from a to v visiting some SCCs has its PRS to be $F \cup e(r)$, where F is a finite set, e is an integer and r is the gcd of the virtual cycle lengths of the visited SCCs. If the path does not pass by any SCC then its PRS contains only the length of the path.*

Lemma 6 *For each vertex v in an a-graph, if $|RS_a(v)| = \infty$ then it can be expressed as follow:*

$$RS_a(v) = F \cup e_1(r_1) \cup \ldots \cup e_k(r_k),$$

for some $k \geq 1$, where F is a finite set, e_i is the start recurring point of the virtual cycle with length r_i.

4 ALGORITHMS

The general steps of the method are:

1. Construct the Adjacency Structure for the graph by navigating the relation B starting at the query constant a to get a connected component of the relation B (CCB), *i.e.* to get the a-graph.
2. Compute the linking points of the bridges in the B-relation (LPB) and linking points of the bridge in the C-relation (LPC). (Here, we use projection and join operators.)
   $LPB := CCB \cap \prod_1 A$; {or $LPB := \prod_1(CCB \bowtie A)$}
   If $LPB = \emptyset$ then do print ("no answer for this query") ; terminate ; end
   else do $LPC := \prod_2 (LPB \bowtie A) \cap \prod_1 C$;
   If $LPC = \emptyset$ then do print ("no answer for this query") ; terminate ; end ;
3. Apply DFS to find the SCCs in the a-graph and decide the pivots. For each SCC, enumerate all the cycles. (This step could be precomputed in a compile phase.)
4. Find all pivots as well as joints in the graph and compute their corresponding self-recurrence sequences. (This step could also be precomputed in a compile phase.)
5. For each b in LPB, compute $RS_a(b)$ and for each bridge (b,q) in relation A, apply steps 1, 3, and 4 to figure out the q-graph.
6. For each $v \in$ q-graph, compute $RS_q(v)$, and
7. Find the intersection of $RS_a(b) \cap RS_q(v)$ to decide if v is an answer.

4.1 Complexity of the Algorithm

We give here only a rough description. For a full treatment, see (Wu 1988). Let n and e be the numbers of vertices and edges in a graph, respectively. Clearly, the computation time needed in steps 1 and 2 is in O(n+e) (Horowitz and Sahni 1978). In step 3, finding the SCCs requires O(n+e) (Tarjan 1973). Within an i-th SCC, to enumerate all the simple cycle needs time in $O((n_i+e_i)c_i)$ (Johnson 1975), where n_i and e_i are numbers of vertices and edges, respectively and c_i is the number of cycles in the i-th SCC (so, $\sum n_i \leq n$ and $\sum e_i \leq e$.) Although such enumeration is proceeded within an SCC, there might be cases suffering the combinatorial explosion. Fortunately, this step can be preprocessed. In step 4, the major purpose is to decide what vertices are pivots and/or joints as well as to compute their SRSs. In an i-th SCC the construction of cycle trees takes $O(n_i)$, provided that the computation of gcd and lcm (Aho et al. 1974) is in O(1). So, the step is in $O(n^2)$.

In step 5, Computing $RS_a(b)$ through the enumeration of simple paths as stated before, may encounter the combinatorial explosion. Alternately, we can compute the RSs of all vertices starting from the origin (it is done by deleting the backward edges after an DFS traversal.) Since the space of a cycle tree can be regarded as a constant relatively to the total number of vertices in the graphs, so the merging can be done in a constant time. Thus, computing the RSs requires O(ne). For each q-graph, the computation steps are the same. So, step 6 is also in O(ne). Note that we never distinguish the parameters in the a-graph with the q-graph. The main purpose in the step 7 is to decide whether or not each vertex w in one of the q-graphs is an answer. Since according to the Cycle Intersection Theorem deciding the emptiness of the intersection of two cycles needs only constant time, so the time to decide if the intersection of $RS_a(b)$ and $RS_q(w)$ is empty (in the worst case) can also be regarded as constant. Thus, in step 7 deciding the answers in a q-graph is in O(n), where n is the number of vertices in a q-graph. Therefore, the total time to process a linear recursive query is dominated by the RS commutation and hence, it is in O(ne), where e is the total number of accessed tuples in the base relations and n is the total number of vertices in the corresponding connected graphs.

5 CONCLUSION

We have described a method in which preprocessing the data with a relatively low cost for maintenance. Since all pivots except the query constant and joints are invariant to any query constant, their SRSs can be calculated and stored during a data preprocessing phase. At query evaluation time, the RS is computed from the information of SRSs. To reduce maintenance effort, the cycles can be partitioned into two classes: one is called useful cycles that can really contribute some occurrences to some of the SRSs and the other is called useless cycles that can not. Maintaining only these useful cycles in a data base does really save a great amount of effort. Recomputing the SRSs is required only when an inserted data produces a new cycle with its length relatively prime to the length of the virtual cycle or when a deleted data causes some useful cycle is broken. Since the virtual cycles of pivots and/or joints in the strongly connected component are of the same length, the difference among them is only the starting recurring value and some finite distinct values. Thus, the final data structure should be reducible to a small amount of space and the time for path merging also reduced.

REFERENCES

(Aho et al. 1974) A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms, Addison-Wesley Pub. Co., June, 1974.*

(Bancilhon et al. 1986) F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

(Bancilhon and Ramakrishnan 1986) F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies,"*Preprints of Workshop on Foundations of Deductive Databases and Logic Programming,Washington, D.C., August 1986.*

(Gallaire et al. 1984) H. Gallaire, J. Minker and J.M. Nicolas, "Evaluation of Data Bases: A Deductive Approach," *Computing Survey, Vol. 16, No. 2, June 1984.*

(Haddad and Naughton 1988) R.W. Haddad and J.F. Naughton, "Counting Method for Cyclic Relations," unpublished manuscript, March 1988.

(Han and Henschen 1987) J. Han and L.J. Henschen, "Processing Linear Recursive Database Queries by Level and Cycle Merging," *Northwestern University EECS Technical Report 87-05-DBM-01.*

(Han and Henschen 1986a) J. Han and L.J. Henschen, "Handling Redundancy in the Processing of Recursive Database Queries," *Northwestern University EECS Technical Report 86-09-DBM-03.*

(Han and Henschen 1986b) J. Han and L.J. Henschen, "Compiling and Processing Transitive Closure Queries in Relational Database Systems," *Northwestern University EECS Technical Report 86-06-DBM-02.*

(Henschen and Naqvi 1984) L.J. Henschen and S. Naqvi, "On compiling Queries in Recursive First-Order Data Bases," *JACM, Vol .31, January 1984, pp.47-85.*

(Horowitz and Sahni 1978) E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms, Computer Science Press, 1978.*

(Johnson 1975) D.B. Johnson, "Finding All the Elementary Circuits of a Directed Graph," *SIAM J. Comput.,Vol.4, No.1, March 1975, pp.77-84.*

(Niven and Zuckerman 1966) I. Niven and H. Zuckerman, *An Introduction to the Theory of Numbers, Second Edition, John Wiley & Sons, Inc. New York, March 1966, pp. 3-7.*

(Mateti, and Deo 1976) P. Mateti, and N. Deo, "On Algorithms for Enumerating all Circuits for a Graph," *SIAM J. Comput., Vol. 5, no.1, March 1976, pp.90-99.*

(Tarjan 1973) R.E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput., Vol. 2, No. 3, September 1973, pp.221-216.*

(Wu 1988) C.S. Wu, "An Algorithmic Approach for Handling Cyclic and Non-cyclic Linear Recursive Queries in Horn Databases," *Ph.D. Dissertation, EECS, Northwestern University, March 1988.*