Vol. 37 No. 12 Transactions of Information Processing Society of Japan Dec. 1996

Regular Paper

Efficient Evaluation of One-directional Cycle-recursive Formulas

Xiaoyong Du,[†] Zhibin Liu[†] and Naohiro Ishii[†]

Normalization is an efficient method of optimization and evaluation for linear recursions. It facilitates the generation of efficient execution plans, since it can directly select the most efficient evaluation algorithm from a set of candidates according to the structural characteristics of recursions. Therefore, it is useful to develop new specialized algorithms for some frequently appearing classes of linear recursions in the normalization framework. This paper focuses on a class of linear recursions, which in terms of the graph model[11] are called one-directional cycle recursions. We show that one-directional cycle recursion is a very frequently appearing pattern in the normalization of linear recursions. We also prove that this kind of recursion can be normalized to a specially formed formula in which all chains have same chain predicate. Based on this interesting property, an efficient evaluation algorithm is proposed. It is efficient because it evaluates only one binary transitive closure and does not need to trace initial driver information in evaluation of the transitive closure, as it is necessary in the multi-way counting method[7]. Some simulation results also show the efficiency of our method.

1. Introduction

Normalization[5],[7],[8],[10],[11] is an optimization method for linear recursions. It transforms linear recursive formulas into normally formed formulas as follows:

$$P(x_1,\cdots,x_n) :- A_1(x_1,y_1),\cdots, A_n(x_n,y_n), P(y_1,\cdots,y_n) \qquad (1)$$

It is also called *n-chain recursion*. Each predicate $A_i(x_i,y_i)$ $(i=1,\cdots,n)$ is called a *chain predicate*.

One important advantage of normalization is that it facilitates the generation of efficient execution plans based on deep analysis of definition of recursion. The compiler can select an appropriate evaluation algorithm directly from a set of candidates. For example, a specialized efficient algorithm can be applied to recursions that can be compiled into bounded or single-chain recursions[6]. It is clearly more efficient than general linear recursion algorithms, such as Semi-Naive algorithm[1]. In this framework, therefore, it is meaningful to develop some efficient algorithms for special classes of recursions, especially if these classes of recursions appear frequently. However, besides bounded recursions, single-chain recursions, and two-chain recursions, there are less efficient algorithms for other classes of recursions. Hence, it is still necessary to develop some efficient new algorithms for those frequently appearing special classes of recursions.

In this paper, we focus on a class of frequently appearing linear recursions called one-directional cycle recursions[11]. By using a graph model[11], a linear recursion can be represented by a hybrid graph, which consists of several connected components. By appropriate graph transformation operations, a connected graph can always be transformed into a one-directional cycle[5]. This means that one-directional cycle recursions are a very frequently appearing pattern if we consider the process of normalization; therefore, a corresponding efficient algorithm is meaningful.

On the other hands, let us consider how to evaluate normally formed formulas. The Counting method[2] is recognized as one of the most efficient algorithms[1] for linear recursive query processing. The main reason for this is that it reduces the interaction of EDB relations on different chains of a recursion by registering the relative distances (levels) from query constants. Essentially, it transforms the processing of a query on a two-chain recursion into the processing of two single-chain recursions (transitive closures), so as to make many optimization techniques for transitive closures applicable to two-chain recursions. Unfortunately, this does not work for recursions in which there are more than three chains, and the performance becomes very low, since it implicitly contains a cross-product if we use the Counting method. Hence the Counting method is essentially suit-

† Department of Intelligence and Computer Science, Nagoya Institute of Technology

able only for simple two-chain recursions.

The multi-way counting method [7] is a generalized counting method for evaluating n-chain recursions. It first decides an appropriate processing direction for each chain. All chains are divided into up chains and down chains by a quad-state variable binding analysis technique. Then, appropriate processing algorithms are chosen for the evaluation of every chain. However, since two-chain recursions have no the properties, as mentioned above, it is necessary to trace not only synchronization information but also the initial drivers for all down-chains. That means that all intermediate relations contain at least four arguments, *id*, *level*, x, y, where *id* is the identifier of the deriver in the exit relation, *level* is the distance between x and y, x is the distinguished variable in a chain furthest from the exit, and y is the variable in the same chain nearest to the exit. Registering the *id* and *level* for each tuple clearly imposes a heavy burden on the processing, and ruins the performance, since it is known that the arity of the intermediate relations strongly influences the performance [1]. It is thus necessary to develop an algorithm that can avoid the problem for some special classes of recursions.

In this paper, we find some important properties of one-directional cycle recursions. A one-directional cycle recursion can be normalized as a special n-chain recursion, in which each chain predicate is one of a cyclic permutation of n initial predicates. This n-chain recursion can be further transformed into a special n-chain recursion in which all chains have the same chain predicate. Based on these properties, an efficient algorithm is proposed for evaluating n-chain recursions of this kind. The algorithm uses only one binary transitive closure and some nonrecursive processing. Furthermore, it does not need to register the information on the initial drivers in evaluating the transitive closure.

This paper is organized as follows: In Section 2, we briefly introduce a graph model defined for single linear recursions [5]. Then, in Section 3, we show that one-directional cycle recursions are a frequently appearing pattern in the normalization process of linear recursion. We also prove two properties of one-directional cycle recursions. On the basis of these properties, we propose a new algorithm in Section 4 for evaluating such normally formed formulas. The basic theoretical analysis and simulation results in Section 5 show that the algorithm is efficient.

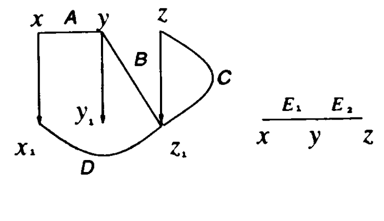


Fig. 1 IE-graph.

Our conclusions are presented in Section 6.

2. Graph Model

In this section, we briefly introduce a graph model that is an extension of the I-graph in Yong, et al. [11]. A detailed definition can be found in Du, et al. [5].

Consider the following linear recursion:

$$\begin{aligned} r_1 &: P(x_1, \cdots, x_n) : -E(x_1, \cdots, x_n) \\ r_2 &: P(x_1, \cdots, x_n) : -A_1(y_1, \cdots, y_k), \cdots, \\ &\quad A_m(z_1, \cdots, z_l), P(w_1, \cdots, w_n) \qquad (2) \end{aligned}$$

It can be represented by an IE-graph (G_i, G_e) [5]. The rule r_2 can be represented by a hybrid graph G_i, called an I-graph, in which variables are represented as nodes labeled with variable names, and predicates A_i are represented as undirected hyperedges labeled with predicate names[☆]. Between each pair of nodes x_i and w_i, there is a directed edge labeled P. The rule r_1 can be represented by an undirected graph G_e, called an E-graph. This is similar to an I-graph except that it contains no directed edges. There may be more than one undirected edge between a pair of nodes. For simplicity, they can be replaced with a new edge, labeled by a new label that is the set of all the labels attached to the edge.

Example 2.1: Consider the following recursive formula:

$$\begin{aligned} P(x,y,z) : - &\quad E_1(x,y), E_2(y,z) \\ P(x,y,z) : - &\quad A(x,y), B(y,z_1), C(z,z_1), \\ &\quad D(x_1,z_1), P(x_1,y_1,z_1) \end{aligned}$$

Its IE-graph is shown in **Fig. 1**. □

Traversals are allowed in I-graphs. A traversal may form a cycle. A cycle is called a *one-directional cycle* if all directed edges in the cycle have the same direction; otherwise, it is called a *multi-directional cycle*. A cycle has a weight. Each directed edge is allocated a weight of 1 or

☆ The label should record the order of reference to nodes, since predicates in Datalog are position-sensitive

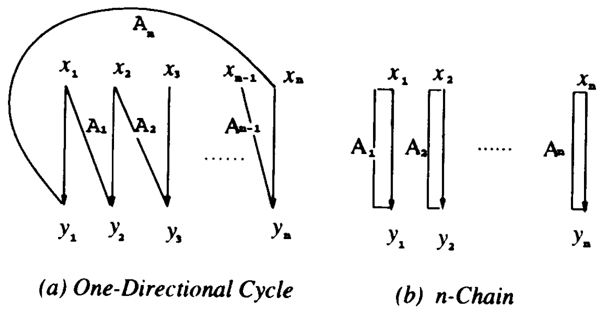


Fig. 2 One-directional cycle.

-1 if it is in or opposite to the direction of the traversal. The weight of an undirected edge is 0. A one-directional cycle with a weight of 1 is called a *unit cycle*. An IE-graph consisting of n unit cycles is called a normal IE-graph or an n-chain.

Figures 2(a) and (b) show IE-graphs of a one-directional cycle and a normal IE-graph, respectively.

A node appearing in the tail of an arrow is called a *C-node*, while a node appearing in the head of an arrow is called an *A-node*☆. They are denoted as V_c, and V_a, respectively. Node x is called *a derived node* of y if x and y appear in the head and tail of a directed edge, respectively. An I-graph with n C-nodes is called an *n-D I-graph*.

Based on the IE-graph representation, the process of normalization can be considered as a process of graph transformation, that is, a process of transforming a general IE-graph into a normal IE-graph.

Obviously, a graph equivalence definition is necessary for graph transformation. Our method is based on a set of generated graphs obtained from the IE-graph.

A set of basic graph operations are proposed for representing the generated graphs. (*1*) *Breeding*, denoted as $\beta_H(G)$, is a unary operation on graph G to generate a new graph as follows: Let x be a C-node in I-graph H, and let y be the derived node of x in H. If x appears in G, then replace it with y. All other nodes in G are replaced with new nodes. (*2*) *Gluing*, denoted as $G * H$, is a binary operation that merges all identical nodes, and replaces two directed edges $P\langle x, x_1\rangle$ and $P\langle x_1, x_2\rangle$ with a new directed edge $P\langle x, x_2\rangle$. Moreover, if G or H is a set of disjoint graphs, then $G*H = \{x*y | x \in G, y \in H\}$. (*3*) *Generating*, denoted as $gen(H)$, is a unary operation on an I-graph H, which eliminates all directed edges from H to form an undirected graph.

☆ There may be some nodes that are neither C-nodes nor A-nodes. Such "intermediate" nodes can be eliminated by join and projection in relational terms. Therefore, we consider only C-nodes and A-nodes here.

Definition 2.2: Let $G(r) = \langle G_i, G_e\rangle$ be an IE-graph. A generated graph of G, denoted as R_G, is a set of undirected graphs (E-graph) defined as follows:

$$\begin{cases} R^k(r) = gen(G_i^k * \beta_{G_i^k}(G_e)) \ (k = 1, \cdots) \\ R^0(r) = G_e \end{cases}$$

where $G_i^k = G_i^{k-1} * \beta_{G_i^{k-1}}(G_i)$, and $G_i^1 = G_i$.

Since an E-graph corresponds to a conjunctive formula, the equivalence of two generated graphs can be defined by the equivalence of the corresponding conjunctive formulas.

Definition 2.3: Let G and H be two E-graph representations of conjunctive formula r and s, respectively. G = H if and only if the two formulas r and s have same results. Let R_1 and R_2 be two sets of E-graphs. $R_1 = R_2$ if and only if there is a one-to-one correspondence f between R_1 and R_2 and for each $x \in R_1$, there is $x = f(x)$.

Definition 2.4: Let $G(r_1) = \langle G_i, G_e\rangle$ and $H(r_2) = \langle H_i, H_e\rangle$ be two IE-graphs defined for two linear recursive formulas r_1 and r_2 respectively. Let R_G and R_H be the generated graphs of G and H, respectively. G and H have strong equivalence, denoted as $G = H$, if $R_G = R_H$. G and H have weak equivalence, denoted as $G \approx H$, if $R_G = A \cup B * R_H$, where A and B are two undirected graphs and $\cup$ is the set union operation.

Two strongly equivalent IE-graphs will produce same results, while two weakly equivalent IE-graphs will not. However, if two IE-graphs are weakly equivalent, we can derive the result of one IE-graph by some nonrecursive operation, such as join and union, on the result of the other IE-graph. Strong equivalence is basic, but weak equivalence is still meaningful in optimization, as we show in the next section.

3. One-directional Cycle Recursions

In this section, after showing that one-directional cycle recursions are frequently appearing patterns in normalization, we study their properties.

3.1 Normalization Based on Graph Transformations

A general linear recursive formula can be transformed into a normally formed formula by a set of equivalence-preserving graph transformation operations that we proposed in a previous paper[5]. Here we merely introduce the definitions of these operations and theorems relative to normalization, explaining these concepts and theorems by means of examples. Details can be found in the above mentioned paper[5].

3.1.1 Reducing

Definition 3.1: Let $G = \langle G_i, G_e \rangle$ be an IE-graph. Assume that C-nodes x_1 and x_2 are connected by an edge[☆], and that their derived nodes are y_1 and y_2, respectively. They can then be vectorized into a new node X. Through this vectorization, G is transformed into a new IE-graph $H = \langle H_i, H_e \rangle$, called a reduction of G, as follows:

- H_i is obtained by replacing x_1 and x_2 with X, and replacing y_1 and y_2 with Y, where Y is a vector of y_1, and y_2. If there are two identical directed edges $P\langle X, Y \rangle$, they are merged.
- H_e is obtained by replacing x_1 and x_2 with X, and merging identical nodes.

Example 3.2: Consider the following linear recursion:

$$\begin{aligned} &P(x_1, x_2, x_3, x_4) : -E(x_1, x_2, x_3, x_4) \\ &P(x_1, x_2, x_3, x_4) : -A(x_1, y_2), B(x_2, y_3), \\ &\qquad C(x_3, x_4), D(y_3, y_4), F(y_1, y_4), \\ &\qquad P(y_1, y_2, y_3, y_4) \qquad (3) \end{aligned}$$

The corresponding IE-graph is shown in **Fig. 3** (a). It can be converted into (b) by vectorizing x_3 and x_4. □

The I-graph in Fig. 3 (b) contains a one-directional cycle component. In fact, we have the following theorem:

Theorem 3.3: A connected general I-graph can be transformed into a simple I-graph by finite reducing operations, where simple I-graph is an I-graph in which there is no connection between any two C-nodes x and y.[5]

This theorem means that a connected I-graph contains at most a one-directional cycle that may be connected by some detachable nodes[4),5)] after reduction. These detachable nodes can be eliminated by the following realigning operation:

☆ Two nodes x and y are connected if there is an undirected edge between them or if there is another node z such that x and z are on an undirected edge and z and y are connected.

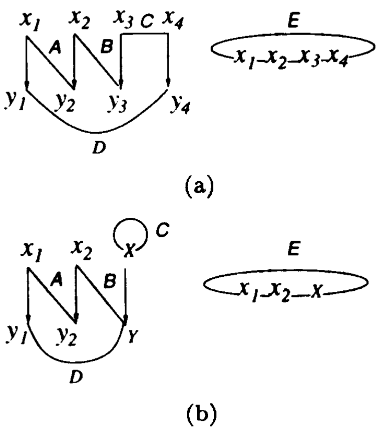


Fig. 3 Reducing.

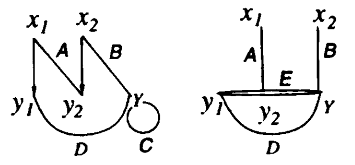


Fig. 4 Realigning C(X) in Fig. 3 (b).

3.1.2 Realigning

Definition 3.4: Let $G = \langle G_i, G_e \rangle$ be an IE-graph of a linear recursive formula r, and let K be a subgraph of G_i satisfying the condition that if A-node x_1 is in K, then $x \in K$ and all edges that contain node x are in K, where x_1 is the derived node of x. Realigning K in G means transforming G into $H = \langle H_i, H_e \rangle$ as follows:

- H_i is obtained by replacing every edge $C(x, y) \in K$ in G_i with $C(x_1, y_1)$, where x_1 and y_1 are derived variables of x and y, respectively. If x or y is not connected to any other nodes, then it is also eliminated.
- H_e is obtained by eliminating all edges $C(x, y) \in K$ from $gen(G_i * \beta_{G_i}(G_e))$.

Realigning is a weak equivalence-preserving operation.

Example 3.5: The C-node X in Fig. 3 (b) can be eliminated by realigning the edge $C(X)$. The resulting IE-graph is shown in **Fig. 4**. □

Please note that Fig. 4 is a pure 2-arity one-directional cycle. As stated in Section 2, all the connected undirected edges between two nodes can be replaced by a new edge. For example, the IE-graph in Fig. 4 can be simplified to obtain a new IE-graph, shown in **Fig. 5** (a), where F and E_1 represent sets of labels in the original graph.

Realigning can detach all those elements that

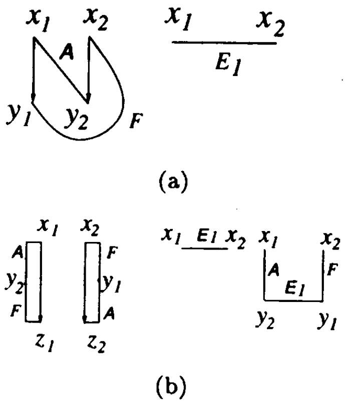


Fig. 5 2-expanding operation.

are not in the cycle, so as to transform the simple IE-graph into a pure one-directional cycle.

3.1.3 Expanding

Definition 3.6: Let $G \doteq \langle G_i, G_e \rangle$ be an IE-graph of a linear recursive formula r. k-expanding is a graph transformation operation that converts G into $H = \langle H_i, H_e \rangle$ as follows:

$$H_i = G_i^k;$$

$$H_e = G_e \cup gen(G_i^1 * \beta_{G_i^1}(G_e)) \cup \cdots$$
$$\cup\, gen(G_i^{k-1} * \beta_{G_i^{k-1}}(G_e))$$

Clearly, the 1-expanding of G_i is itself.

Example 3.7: The IE-graph in Fig. 5 (a) can be converted into Fig. 5 (b) by 2-expanding. Figure 5 (b) is a pure 2-chain recursion. □

An expanding operation transforms a one-directional cycle into a normal IE-graph.

The above discussion shows that all connected general IE-graphs can always be transformed into one-directional cycles by applying reducing and realigning operations. The one-directional cycle can be further transformed into a normal IE-graph by expanding. Hence we can show that a one-directional cycle is a frequently appearing pattern of an IE-graph in normalization.

3.2 Properties of One-directional Cycle Recursions

We further study the properties of one-directional cycle recursions, which are the basis of our evaluation method.

Theorem 3.8: Consider a one-directional n-weighted cycle recursions:

$$P(x_1, \cdots, x_n) : -A_1(x_1, y_2), \cdots, A_n(x_n, y_1),$$
$$P(y_1, \cdots, y_n) \qquad (4)$$

Its IE-graph is shown in Fig. 2 (a).

(1) It can always be transformed into an n-chain recursion by an n-expanding operation on its corresponding IE-graph.

(2) The n-chain recursions have the following form:

$$P(x_1, \cdots, x_n) : -f_1(x_1, y_1), \cdots, f_n(x_n, y_n),$$
$$P(y_1, \cdots, y_n) \qquad (5)$$

where $f_1, \cdots, f_n$ are cyclic permutations of the list of initial predicates $A_1, \cdots, A_n$.

The first part of this theorem is proved in Yong et al.[11], and the second part can be derived directly from its formalization process.

Example 3.9: The one-directional recursion in Example 3.7 is transformed into a two-chain recursion after a 2-expanding operation. Its two chain predicates are AF and FA, which are two cyclic permutations of predicates $\{A, F\}$, respectively (see Fig. 5). □

Theorem 3.10: n-chain recursion (5) is weakly equivalent to a special n-chain recursion called the *same-chain normally formed formula* (SCNF for short), in which all chain predicates are the same:

$$Q(x_1, \cdots, x_n) : -f_1(x_1, y_1), \cdots, f_1(x_n, y_n),$$
$$Q(y_1, y_2, \cdots, y_n) \qquad (6)$$

Proof: We show how to construct such an SCNF recursion from formula (5). Let all the generation graphs of an n-chain recursion (5) be $R_0, R_1, \cdots, R_m$. Consider the generation graph R_i ($i > 0$). It can be divided into three parts: an exit-part, an extra-part, and a loop-part (see **Fig. 6**). The extra-part consists of n disconnected edges h_j, which are generated from edges f_j ($j = 1, \cdots, n$). Edge h_j is labeled with $A_j, \cdots, A_n$ predicates ($j = 1, \cdots, n$). Correspondingly, the exit-part consists of n edges,

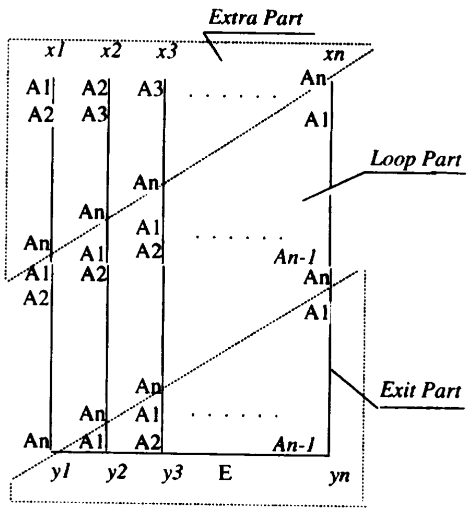


Fig. 6 Generation graph R_2.

$e_1, \cdots, e_n$, connected by E edges, where e_j is labeled with predicates $A_1, \cdots, A_j$ in f_j ($j = 2, \cdots, n$) and e_1 is an empty set. The loop part contains n chains, each of which consists of a set of connected edges labeled with f_1.

Let Q_i denote the subgraph of R_{i+1} when the extra-part is deleted. Clearly, the set $\{Q_0, Q_1, \cdots, Q_{m-1}\}$ is exactly a set of generation graphs of an SCNF recursion, where the exit-part in R_i is the exit predicate of the SCNF recursion. Since the extra-part is a non-recursive part, from the definition of weak equivalence, this SCNF recursion is weakly equivalent to the original recursion formula (5).

Corollary 3.11: If the chain predicates of an n-chain recursion are a subset of cyclic permutations of a set of initial predicates $\{A_1, A_2, \cdots, A_m\}$, then the recursion can also be transformed into an SCNF.

According to Theorems 3.8 and 3.10, it is possible to obtain a two-step method for evaluating a one-directional cycle recursion. That is, we can first evaluate a corresponding SNCF recursion, and then obtain the final result by a set of nonrecursive join operations and union operations. Although SCNF recursions can be evaluated by general methods, such as the Counting method, Semi-Naive method, they can be evaluated more efficiently by considering the properties that all chain predicates in an SCNF are the same. We propose such an algorithm in the next section.

4. Query Processing Strategy

In this section, we propose a processing strategy for a normally formed formula normalized from one-directional cycle recursions. The strategy consists of two steps: query analysis and SCNF evaluation. The query analysis method is tacken from Han[7]. Its purpose is to decide a suitable entry point of the process so as to reduce the relative data set for recursion evaluation. The evaluation algorithm for SCNFs is the key to this processing strategy. It first transforms a general n-chain recursion into an SCNF recursion according to the theorems in Section 3, and then evaluates an SCNF by means of a specialized algorithm. This algorithm contains only one transitive closure and some non-recursive processing. As in the Counting method, only synchronization information should be traced in the evaluation of the transitive closure. Some post-processing is necessary to obtain the final answer to the query.

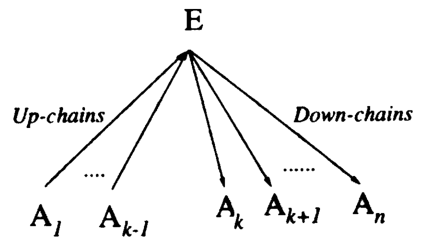


Fig. 7 Processing direction of chains.

4.1 Query Analysis

The purpose of the query analysis step is to divide all chains into two classes: up-chains (driving chains) and down-chains (driven chains). A chain is an up-chain if the chain predicate is restricted by some highly selective query constants. Otherwise, it is a down-chain. Up-chains should be processed first, starting at the end of the chain furthest from the exit, and forming some selection conditions on the exit. Down-chains should be processed from the end connected to the exit down along the chain to the far end (**Fig. 7**).

The simplest type of query analysis is **two-state variable binding analysis**, which is used in the Counting method[2]. The only possible states for any distinguished variable are "bound" and "free". Chains that contain bounded distinguished variables are treated as up-chains, while others are down-chains. This strategy is based on the assumption that a bounded variable is highly selective, and on a well-known heuristic that selection should be performed first. However, this assumption and heuristic are not always true.

Therefore, a more precious **"quad-state variable binding analysis"** method is proposed in Han[7]. A distinguished variable relating to a query can be characterized by two essential notions: instantiation and inquiry. For example, if a variable is constrained as a constant in a query, then it is an instantiated but not necessarily an inquired variable, related to the query.

Furthermore, after the operations of reducing and expanding, the distinguished variables of a chain in an n-chain recursion may consist of more than one initial distinguished variable, or may come from the composition of some initial distinguished variables. Therefore, a set of state folding rules are necessary; such a set is also given in Han[7].

4.2 An Evaluation Algorithm

The n-chain recursion queries can be evaluated by means of the multi-way counting method[7], which first processes all up-chains, and then all down-chains. However, there are two kinds of information that should be traced in the processing of each chain:

1. The synchronization of all chains.
2. The initial driver of each result in the transitive closure of down-chains.

As we stated in the introduction, this imposes heaven burden on the processing. Therefore, we propose a new algorithm that needs only to evaluate one transitive closure without tracing any initial drivers.

Our algorithm is applicable only to one-directional cycle recursions. The algorithm consists of two parts: SCNF evaluation and post-process. The post-process part consists of a set of nonrecursive join operations and union operations. These can be evaluated directly by means of an existing query processor.

If some chains are up-chains - that is, if they have the role of conveying query-bounding information to the exit relation - then they can be processed as in the Counting method, and form a restriction on the exit relation. We can replace the exit relation, together with some restriction from the up-chains, with a new exit relation. Furthermore, from Corollary 3.11, the remaining down-chains can still be transformed into SCNFs. In other words, the following algorithm for SCNF evaluation is available.

In this algorithm, to simplify description, we assume without loss of generality, that all n chains are down-chains.

Algorithm 4.1 (Evaluation of one-directional cycle recursions):

Input: An n-arity one-directional cycle recursion formula (4)

Output: Result of the recursive predicate P

Method:

Step 1: Evaluating the new exit relation, denoted as E_{new}.

$$E_{new}(x_1,\cdots,x_n) = \left\{\begin{array}{r} A_1(x_2,y_2) \\ A_1A_2(x_3,y_3) \\ \cdots \\ A_1\cdots A_{n-1}(x_n,y_n) \end{array}\right\} E(x_1,\cdots,y_n)$$

Step 2: Evaluating the distinctive values that appear in E_{new}, denoted as $V_d(E_{new})$.

$V_d(E_{new}) = \cup_{i=1}^{n}(\Pi_{x_i}(E_{new}(x_1,\cdots,x_n)))$

Step 3: Evaluating the transitive closure of f_1, denoted as C_f, with the initial driver set $V_d(E_{new})$.

$C_f(x,y,i) = f_1^*(V_d(E_{new}))$

where $x \in V_d(E_{new})$, y is the end of f_1 furthest from the exit, and i is the distance between x and y in f_1.

Step 4: Evaluating the SCNF.

$$R(y_1,\cdots,y_n) = E_{new}(x_1,\cdots,x_n), C_f(x_1,y_1,i),\cdots,C_f(x_n,y_n,i)$$

Step 5: Post-processing.

$$P(x_1,x_2,\cdots,x_n) = \left\{\begin{array}{r} A_1A_2\cdots A_n(x_1,y_1) \\ A_2\cdots A_n(x_2,y_2) \\ \cdots \\ A_n(x_n,y_n) \end{array}\right\} R(y_1,\cdots,y_n)$$

5. Performance Analysis

In this section, we first analyze the performance of the algorithm, and then design some experiments to support our analysis.

5.1 Basic Analysis

The following points can be viewed as evidence of the efficiency of our algorithm.

5.1.1 Number of Joins

A large number of join operations may be generated in the evaluation of recursive queries. This is regarded as one of the major reasons for the low efficiency of the evaluation. Therefore, an algorithm that contains fewer join operations can reasonably be considered more efficient.

Here we consider the number of join operations in evaluation of formulas (4) and (5) by multi-way counting and our method, respectively. The two formulas are generated from the same initial formula (2).

For formula (4), there is a preprocess for evaluating all chain predicates f_i $(i = 1,\cdots,n)$ and a new exit predicate, and a counting process for evaluating all transitive closure of f_i $(i = 1,\cdots,n)$ and their join with the exit. Let n be the arity of the recursive predicate, and let m be the average number of iterations in an evaluation of transitive closures. We also assume that some intermediate results are saved for future evaluation. Then there are $n+2n(n-2)+n$ joins in the preprocess, and $mn+n$ joins in the counting process. Thus, the total number of joins is $g_1(m,n) = nm + (2n^2 - n)$.

In evaluation of formula (5) by our method, there is an extra step in the preprocess for evaluating the new exit predicate, and an ex-

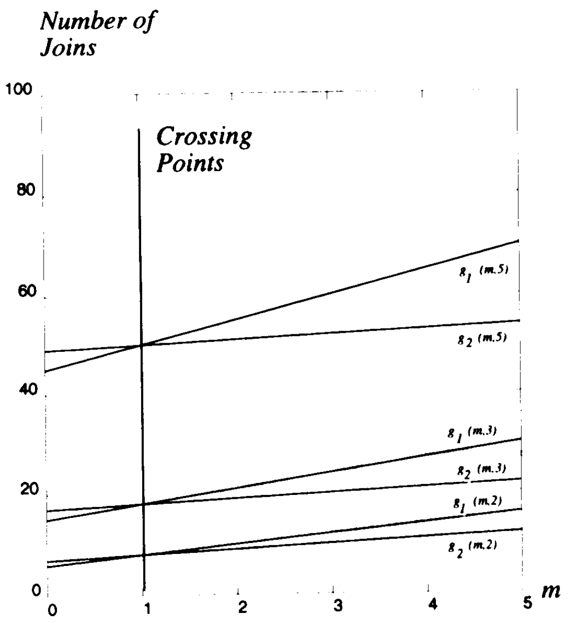


Fig. 8 Comparison of the number of joins.

tra postprocess. The numbers of joins in the three parts are $n + 2n(n-2) + n$, $m + n$, and $n - 1$, respectively. Hence, the total is $g_2(m,n) = m + 2n^2 - 1$.

The lines of $g_1(m,n)$ and $g_2(m,n)$ with $n = 2$, 3, and 5 are shown in **Fig. 8**. From this figure, we have following results:

(1) The crossing point of $g_1(n,m)$ and $g_2(n,m)$ is $m = 1$ if $n \geq 2$. This means that our method is more efficient when more than two times of iterations are necessary for evaluation of the closures of any linear recursions with more than two chains.

(2) The slope of $g_2(n,m)$ becomes larger when n is increased. This means that the larger n is, the more efficient our method is. In other words, the main advantage of our method is that it gives the fewest in 2-chain recursions.

5.1.2 Cost of Evaluating Partial Transitive Closures

It is well known that evaluation of transitive closures is the major part in the process of recursive queries. It is meaningful to analyze the time cost of evaluating partial transitive closures in the two methods.

(1) Bancilhon and Ramekrishnan[1] showed that the size of intermediate relations is an important factor affecting the performance of recursive evaluation. The lower the arity of the intermediate relations, the more efficient the algorithm. It is not necessary to trace the initial driver information when evaluating transitive closures in our method, and consequently the size of intermediate relations is smaller than in the multi-way counting method.

(2) Our method first evaluates the union of all initial derivers, and then evaluates only one partial transitive closure. In contrast, the original method evaluates all the necessary n transitive closures, respectively. It is easy to show that $c(A^*(S)) \ll \sum_{a \in S}(c(A^*(a)))$, where $c(f)$ is the time cost of operation f. Therefore, our method is more efficient.

Besides the above two major aspects, there are some other aspects in which the performance can be improved. For example, since all the side relations in an SCNF are the same, only one copy of that relation needs to reside in memory. This means that there is more free space, which can be used for storing intermediate data and reducing the time spent on paging when evaluating the joins.

5.2 Experiments

Although the above theoretical analysis is important, some experiments are still necessary to confirm that our method has good performance. This subsection explains the design of several experiments for comparing the time cost of our method and the multi-way counting in the 2-arity case. Note that the advantage of our method is lowest in this case, and that the advantage of the multi-way counting method is the highest, because it is reduced and replaced by the ordinary counting method.

Consider a 2-arity one-directional cycle recursion:

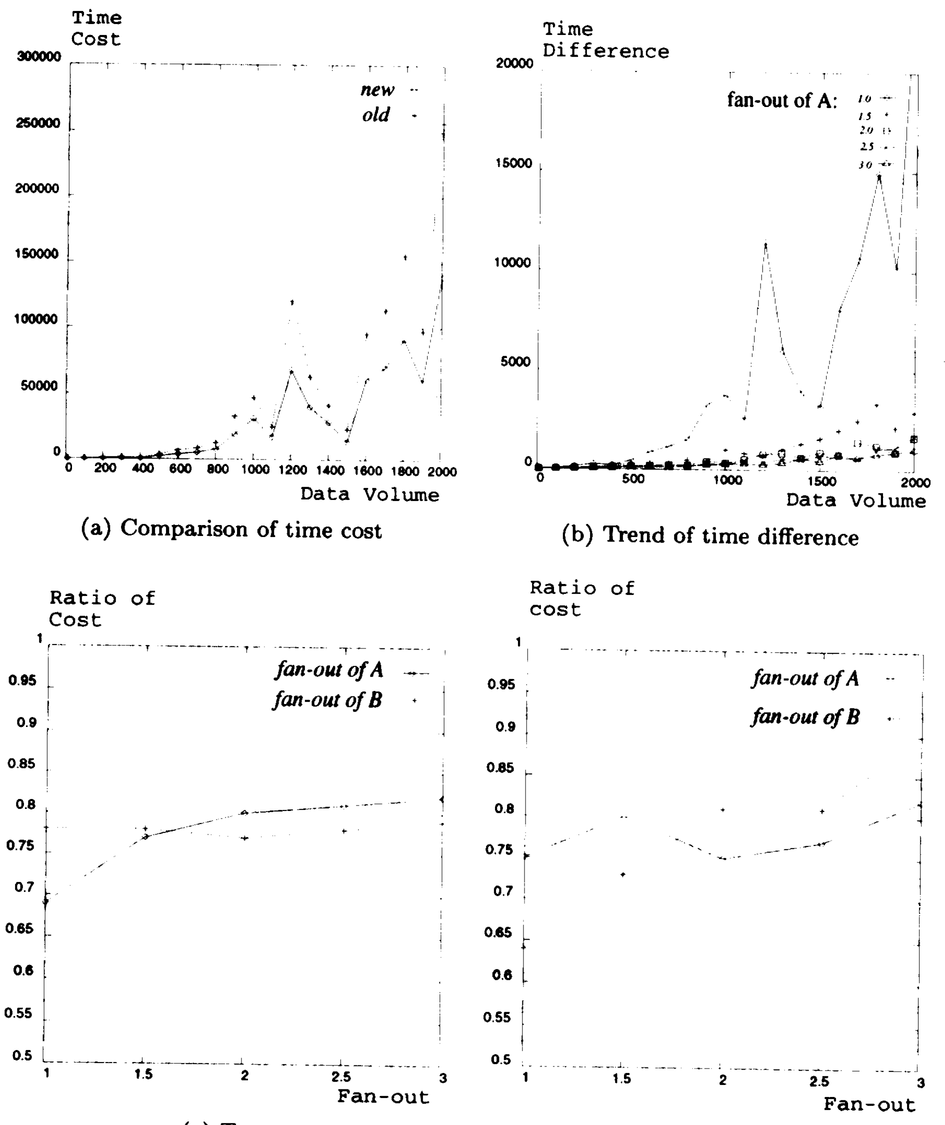
$$\begin{aligned} &P(x,y) : -A(x,y_1), P(x_1,y_1), B(y,x_1) \\ &P(x,y) : -E(x,y) \end{aligned} \quad (7)$$

By means of normalization method, this can be transformed into

$$\begin{aligned} &P(x,y) : -E(x,y) \\ &P(x,y) : -A(x,y_1), E(x_1,y_1), B(y,x_1) \\ &P(x,y) : -C(x,x_1), P(x_1,y_1), D(y,y_1) \end{aligned} \quad (8)$$

By means of our method, it can be further transformed into

$$\begin{aligned} &E_1(x,y) : -E(x,y) \\ &E_1(x,y) : -A(x,y_1), E(x_1,y_1), B(y,x_1) \\ &Q(x,y) : -C(x,x_1), E_1(x_1,y_1), A(y,y_1) \\ &Q(x,y) : -C(x,x_1), Q(x_1,y_1), C(y,y_1) \\ &P(x,y) : -E_1(x,y) \\ &P(x,y) : -Q(x,z), B(y,z) \end{aligned} \quad (9)$$



(a) Comparison of time cost (b) Trend of time difference (c) Tree (d) Reversed Tree

Fig. 9 Experiment results.

where $C(x, y) = A(x, z), B(z, y)$ and $D(x, y) = B(x, z), A(z, y)$.

The Counting method is used to evaluate the formulas (8) and (9).

5.3 Experiment System

The system used for the experiment consists of three layers. It is implemented in C language on Sun Sparc station 5/110 with SunOS 4.1.4. The first layer is a storage subsystem. All available memory units are linked by bidirectional links. Memory units are assigned in groups of 1000 per request. The second layer is a data manipulation subsystem, in which the main data operations are implemented, including join, union, sort, and transitive closure. A Counting algorithm is implemented in the third layer.

5.3.1 Data Generator

We prepare three kinds of data set: tree, reversed tree, and random data set. The tree and reversed tree sets are characterized by two factors: fan-out and size of relation. The fan-out is the average value for a tree. It means that, for each node in the tree, the number of out edges is from 0 to 2 times the fan-out. For example, if the fan-out is 1.5, then the number of out edges is 0 to 3 for the tree. The random data set is characterized by the size of domain and that of the relation.

For simplicity, we specify that the Counting method is only applicable on non-cyclic data sets (trees and reversed trees).

5.3.2 Experiments and Result Analysis

The first experiment is done on tree data sets. The fan-out of base relations A and B are varied from 1.0 to 3.0 by step 0.5, and for each fan-out we vary the size of relations from 100 to 2000 by step 100. Base relation E is generated randomly. All three base relations have the same size in each test.

Figure 9(a) is a typical one in this group of tests. However, the greater the fan-out, the closer the two curves. This is reasonable, since the number of iterations becomes smaller and smaller when the fan-out is increased for the same data volume. This trend is showed in Fig. 9(b).

Figure 9(c) shows the ratio of the time cost of two methods when the fan-out is varied. Our algorithm can improve the performance by 22.2 percent on average. The other interesting result shown in this figure is that the shapes of relations A and B have little effect on the time cost.

There are similar results for reversed trees. Figure 9(d) shows the ratio of the time costs of the two methods. In this case, we vary the ratio of duplication for base relation A and B from 1.0 to 3.0 by step 0.5, and the data volume is varied from 100 to 2000 by step 100.

For a random data set, the Semi-Naive method is used to evaluate formulas (8) and (9). In this case, our method cannot gain any advantage from its transitive closure evaluation. It fact, it does not directly compute directly transitive closures in the Semi-Naive method. However, our method is still efficient when the intermediate result is large enough, because in this case the paging becomes very frequent. Our method requires only one copy of the chain predicate to reside in memory, and thus gains more space for storing intermediate results so as to reduce the number of paging operations.

In the test, we vary the ratio of size of the domain to that of the relation from 1 to 5 by means of step 1, and vary the size of the relation from 100 to 2000 by step 100. **Figure 10** shows that the performance of our method can be improved rapidly when the size of the relation is close to the size of the domain. In this situation, the connectivity between data in the relations is very high, and thus the intermediate results are huge.

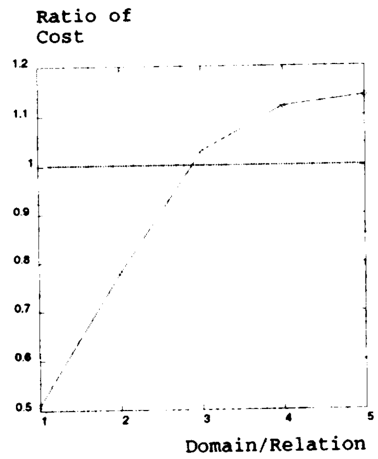


Fig. 10 Random data set.

6. Conclusion

This paper has proposed an efficient algorithm for evaluating a class of linear recursive queries, that is, one-directional cycle recursions. It first transforms a one-directional cycle recursion into a special form called an SCNF, which is an n-chain recursion with the same chain predicate. It then evaluates an SCNF recursion by means of only one binary transitive closure registering only the level information, and executing some non-recursive processing. The importance of the algorithm lies in three aspects: (1) one-directional cycle recursion is a very frequently appearing pattern in normalization of linear recursions, (2) in the framework of normalization, specialized algorithms for some kinds of recursion can be applied directly, and (3) the algorithm is efficient, as is shown by basic theoretical analysis and experiments.

Acknowledgments We would like to thank the anonymous referees and editor, who provided helpful comments and made the paper easy to understand. We also thank Dr. Jeffrey Yu in the Austrlian National University and Dr. Prof. J. Han in Simon Fraser University for helpful discussion and comments.

References

1) Bancilhon, F. and Ramekrishnan, R.: An Amateur's Introduction to Recursive Query Processing Strategies, *Proc. Intl. Conf. ACM SIGMOD*, pp.16–52 (1986).

2) Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J.D.: Magic Set and Other Strange Ways to Implement Logic Programs, *Proc. ACM SIGACT-SIGMOD-SIGART PODS*, pp.1–15 (1986).

3) Du, X. and Ishii, N.: Optimizing Linear Recursive Formulas by Detaching Isolated Variables, *IEICE Trans. Information and Systems* Vol.E78-D, No.5, pp.579–585 (1995).
4) Du, X. and Ishii, N.: Reducing the Arity of Predicates by Realigning Some Predicates, *Proc. ICLP '95 Joint Workshop on Deductive Databases and Logic Programming*. Kanakawa, Japan, pp.57–69 (1995).
5) Du, X. and Ishii, N.: Normalization of Linear Recursions Based on Graph Transformation, *Proc. 5th Intl. Conf. on CISMOD*, Bombay, India, *LNCS*, Vol.1009, pp.265–282 (1995).
6) Han, J.: Selection of Processing Strategies for Different Recursive Queries, *Proc. 3rd Intl. Conf. on Data and Knowledge Bases* (1988).
7) Han, J.: Compiling General Linear Recursions by Variable Connection Graph Analysis, *Comput. Intell.*, Vol.5, pp.12–31 (1989).
8) Han, J. and Zeng, K.: Automatic Generation of Compiled Forms for Linear Recursions, *Information Systems*, Vol.17, No.4, pp.299–322 (1992).
9) Jarke, M. and Koch, J.: Query Optimization in Database Systems, *ACM Comput. Surv.*, Vol.16, No.2 (1984).
10) Lu, W., Lee, D.L. and Han, J.: A Study on the Structure of Linear Recursion, *IEEE Trans. Knowledge and Data Engineering*, Vol.6, No.5, pp.723–737 (1994).
11) Yong, C., Kim, H.J., Henschen, L.J. and Han, J.: Classification and Compilation of Linear Recursive Queries in Deductive Databases, *IEEE Transactions on Knowledge and Data Engineering*, Vol.4, No.1, pp.52–67 (1992).

(Received January 11, 1996)
(Accepted September 12, 1996)

Xiaoyong Du received the B.S. degree in computational mathematics from Hangzhou University, Zhejiang, China in 1983, and the M.E. degree in information and computer science from People's University of China, Beijing, in 1988. From 1989 to 1992, he was a lecturer of the Institute of Data and Knowledge Engineering in People's University of China. He is now a Ph.D. candidate at the Department of Intelligence and Computer Science in Nagoya Institute of Technology, Nagoya, Japan. His current research interests include databases and artificial intelligence. He is a member of IPSJ.

Zhibin Liu received the B.E. degree in computer science and technology from Tianjin University, Tianjin, China in 1985, and M.E. degree in information and computer science from People's University of China, Beijing, China in 1988. From 1990 to 1992 he was a lecturer of the Department of Computer Science and Technology in Tianjin University. From 1992 to 1995, he was a lecturer of the Department of Computer Science and Technology in Beijing Polytechnic University. He is now pursuing the Doctor of Engineering degree in Nagoya Institute of Technology, Nagoya, Japan. His current research interests include algorithm design and analysis, deductive databases, databases integration and heterogeneous database systems.

Naohiro Ishii received the B.E. and M.E. degrees and the Doctor of Engineering degree in Electrical and Communication Engineering from Tohoku University, Sendai, in 1963, 1965 and 1968, respectively. From 1968 to 1974 he was at School of Medicine in Tohoku University, where he worked on information processing in the central nervous system. Since 1975 he has been with the Nagoya Institute of Technology, where he is a professor in the Department of Electrical and Computer Engineering. His current research interests include databases, software engineering, algorithm design and analysis, nonlinear analysis of neural network and artificial intelligence. He is a member of IPSJ, IEICE, ACM and IEEE