NORTH-HOLLAND

*Applications*

# Dynamically Ordered Semi-Naive Evaluation of Recursive Queries

KI-HYUNG HONG*
YOON-JOON LEE

and

KYU-YOUNG WHANG

*Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-Dong, Yusong-Gu, Taejon, 305-701, 807 South Korea*

Communicated by Ahmed K. Elmagarmid

ABSTRACT

Conventional fixed point evaluation techniques evaluate recursions by applying all rules repeatedly using an initial set of tuples (i.e., a given extensional database instance) until no new tuples are generated, but there is no specific order in which rules are applied. We can speed up the evaluation by applying rules in an appropriate order. In this paper, we propose a new fixed point evaluation technique, called the *dynamically ordered semi-naive evaluation* (or simply DYN), in which the next rule to be applied is determined at run time dynamically. DYN consists of a semi-naive algorithm and a set of selection strategies. The semi-naive algorithm allows dynamic ordering of rule applications and makes tuples generated by a rule application immediately available in the subsequent rule applications. After each rule application, the selection strategies determine the next rule by considering the syntactic structure of recursion and some information about the intermediate result up to the present. We develop these selection strategies so that the total number of rule applications and joins can be reduced. Through experimental comparisons, we shows that DYN outperforms the previous evaluation techniques in terms of the total number of rule applications and joins.


*Current address: Database Section, Software Department, Computer Division, Electronics and Telecommunications Research Institute, 161 Kajong-Dong, Yusong-Gu, Taejon, 305-350, South Korea.

*INFORMATION SCIENCES* **96**, 237–269 (1997)

655 Avenue of the Americas, New York, NY 10010

0020-0255/97/$17.00
PII S0020-0255(96)00160-0

## 1. INTRODUCTION

*Datalog* is a logical language based on function-free Horn clause logic. It can be used as a database language and a lot of research has been done [25, 26, 8]. Its expressiveness is more powerful than that of conventional relational database languages. It allows users to compose complex queries, especially those involving recursions. The detailed syntax and semantics of datalog can be found in [9, 25]. Efficient evaluation of recursions is an important issue in processing datalog queries, since it may often require many costly join operations.

The naive evaluation techniques based on Tarski's fixed point theorem [23] evaluate a recursion by applying all the rules of the recursion repeatedly in a loop by using an initial set of tuples until no new tuples are generated. However, the naive evaluation is inefficient because it does not satisfy the following two properties, which are desirable for efficient evaluation of recursions [27, 8, 25].

- *Semi-naive property*: An efficient evaluation technique should not repeat the same computation (or reasoning). The evaluation technique that satisfies this property, usually called the *semi-naive evaluation* technique, involves a phase that rewrites recursive rules into the equivalent ones that do not repeat the same computation by using the differential notation proposed in [1, 2].
- *Relevant-data-only property*: An efficient evaluation technique should not generate tuples irrelevant to the answer for a given query. There have been a lot of research results on this property, such as the magic set and counting/reverse counting methods [3, 5], the marking algorithm [10], the static filtering method [14], and the factoring technique [18]. By propagating the constants given in the query, such techniques rewrite recursive rules in the recursion so as to minimize processing irrelevant tuples to the answer.

Besides the above two properties, the efficient evaluation technique should maximize the effect of each rule application in a loop. The effect of a rule application can be defined as the number of inferences made by the application. Here, an inference is the process that derives a tuple (fact) from a given set of ground facts by applying a rule. Since semi-naive evaluation techniques do not repeat the same inference, they are all equivalent in the total number of inferences made during the evaluation of a recursion for a given extensional database instance [13, 20]. Therefore, making each rule application produce more tuples can reduce the total number of rule applications required in the evaluation of recursions [20].

The fact that more inferences are made by each rule application implies that more inferences can be made using a set of tuples in a page fetched from disk [20]. Thus, we can expect that the total number of I/O operations is reduced. The number of rule applications is closely related to the number of joins, i.e., the reduction of the number of rule applications leads to the reduction of the number of joins. (This is not always true. More details will be given in Section 5.) Furthermore, the reduction of the number of joins (or rule applications) can also reduce the cost due to some fixed overheads associated with each join (or each rule application) [20].

There are two classes of studies to maximize the effect.

- *Immediate utilization of tuples*: In the basic naive/semi-naive evaluation, the tuples produced by a rule application in an iteration can only be used in the next iteration. Making new tuples produced by a rule application immediately available in the subsequent rule applications can speed up the evaluation [17, 7, 16].
- *Selection of an appropriate ordering*: The order of rule applications in the loop is not specified in the basic naive/semi-naive evaluation. However, the order of rule applications may affect the performance of evaluation significantly. There have been some results [20, 16] for finding a good order of rule applications. In this paper, we concentrate on this subject.

Since all the previous techniques set the order of rule applications at compile time, we call them *static ordering* techniques. When they choose an order of rule applications for a given recursion, they consider only the syntactic structure of the recursion which is a set of dependency relationships between recursive rules and recursive predicates in the recursion. However, the optimal ordering that minimizes the total number of rule applications also depends on the content of the extensional database instance given at evaluation time. There is no guarantee that an ordering that is optimal on one extensional database instance is also optimal for another extensional database instance.

In this paper, we propose a new fixed point evaluation technique, called the *dynamically ordered semi-naive evaluation* (or simply DYN), in which the next rule to be applied is determined at run time dynamically. DYN consists of a semi-naive algorithm and a set of selection strategies. The semi-naive algorithm allows dynamic ordering of rule applications and makes tuples generated by a rule application immediately available in the subsequent rule applications. After each rule application, we classify recursive rules into two groups: *active rules*, if there are some tuples that have not been used in their previous applications, and *inactive rules*, otherwise. The selection strategies determine the next rule by considering

the syntactic structure of recursion and some information from the intermediate result up to the present. The information from the intermediate result at a given time is as follows: for each recursive rule, whether it is active or inactive, and for each active rule, the number of rule applications that have been made after its most recent application. Even after the same sequence of rule applications, such information varies according to the content of the extensional database instance used in the evaluation. We develop the selection strategies so that the total number of rule applications and joins can be reduced. Through experimental comparisons, we show that DYN outperforms the previous evaluation techniques in terms of the total number of rule applications and joins.

We do not deal with the relevant-data-only property in this paper. One can achieve this property by applying some rewriting techniques that deal with this property such as the magic set method [3] to a given recursion before using the technique proposed in this paper.

This paper is organized as follows. In the next section, we explain our motivation and summarize the related work. In Section 3, we introduce the terminology used in this paper. In Section 4, we present an algorithm that allows the dynamic ordering of rule applications. In Section 5, we present the selection strategies. In Section 6, we compare performances between our technique and the previous techniques through experiments. Finally, we conclude the paper in Section 7.

## 2. RELATED WORK AND MOTIVATION

### 2.1. *RELATED WORK*

Let us consider the datalog program $\mathscr{P}_1$, shown in Figure 1. $\mathscr{P}_1$ is a version of a nonlinear same generation program rewritten using the supplementary magic set technique [5]. Here, *up*, *down*, and *flat* are extensional predicates.

Consider the basic naive/semi-naive evaluation [4] that does not satisfy the immediate utilization of tuples. The evaluation of $\mathscr{P}_1$ begins with the application of $r_1$ which is the exit rule of the recursion having the seven recursive rules, $r_2, r_3, \ldots, r_8$. After applying the exit rule, the seven recursive rules should be applied repeatedly in a loop. We assume that, in the loop, the order of rule applications for $\mathscr{P}_1$ is $O_1$, $(r_2, r_3, r_4, r_5, r_6, r_7, r_8)$. Before starting the loop, the relations for all the recursive predicates except *msg* are empty. The application of the exit rule $r_1$ produces a new tuple (1) for *msg*. In the first iteration, only two applications of $r_2$ and $r_5$ are meaningful, i.e., only the application of $r_2$ and $r_5$ may generate some

$(\mathcal{P}_1)$ $r_1$ : $msg(1)$.

$r_2$ : $supm2(X,Y) \leftarrow msg(X), up(X,Y)$.

$r_3$ : $supm3(X,Y) \leftarrow supm2(X,Z), sg(Z,Y)$.

$r_4$ : $supm4(X,Y) \leftarrow supm3(X,Z), flat(Z,Y)$.

$r_5$ : $sg(X,Y) \leftarrow msg(X), flat(X,Y)$.

$r_6$ : $sg(X,Y) \leftarrow supm4(X,Z), sg(Z,W), down(Z,Y)$.

$r_7$ : $msg(X) \leftarrow supm2(Z,X)$.

$r_8$ : $msg(X) \leftarrow supm4(Z,X)$.

$r_9$ : $query(Y) \leftarrow sg(1,Y)$.

Fig. 1. $\mathcal{P}_1$.

tuples since *msg* is not empty. The applications of other rules in the first iteration are meaningless since they include joins (or projections) with empty relations. The application of $r_7$ becomes meaningful in the second iteration if the application of $r_2$ in the first iteration generates some new tuples for *supm*2. Furthermore, the application of $r_8$ remains meaningless at least until the fourth iteration.

The evaluation techniques satisfying immediate utilization of tuples were independently developed by Ceri et al. [7], Kildall [15], Cai and Paige [6], and Lu [17]. The technique proposed by Ceri et al. [7] is the naive evaluation technique based on immediate utilization of tuples and is identical to the basic naive/semi-naive evaluation in that an iteration consists of applying all the recursive rules in a given recursion. However, in this technique, each rule application in an iteration uses the results of previous rule applications in the same iteration. For example, under the order $O_1$, the application of rule $r_7$ in the first iteration is meaningful if the application of $r_2$ generates some tuples for *sump*2. If all the meaningful applications produce some tuples, then the application of $r_8$ becomes meaningful in the second iteration.

Kildall [15] and Cai and Paige [6] proposed iterative algorithms to evaluate systems of fixed point equations. Their algorithms iterate each equation individually. That is, each iteration computes only one fixed point equation. Therefore, the result from the computation of an equation is

used immediately in the subsequent computation (or iteration). However, they do not address the ordering of rule applications. In the context of transitive closure, Lu [17] proposed a technique that could utilize tuples immediately in the same iteration in which they were generated. Even though the transitive closure for a given relation is a simple recursion with only one recursive rule, Lu's technique applies the rule multiple times in an iteration. Each rule application in an iteration uses a subset of the relation, and the other rule applications in the same iteration use the resulting tuples with the remaining part of the relation.

According to the order defined above, only applications of $r_2$, $r_5$, and $r_7$ are meaningful in the first iteration, even if we use the technique based on immediate utilization of tuples and assume that all the meaningful applications produce some new tuples. Consider a different order $O_2$, $(r_2, r_7, r_5, r_6, r_3, r_4, r_8)$. In this order, the applications of all the recursive rules except $r_6$ are meaningful in the first iteration under the same assumption. The application of $r_6$ in the first iteration includes a join with empty relation because $r_6$ is applied before $r_4$ that is the only rule with *supm*4 as its head. This indicates that ordering rule applications may significantly affect the performance of evaluation. There have been a few research results [20, 16] on finding a good order of rule applications.

Ramakrishnan et al. [20] presented a theoretical analysis of rule application orderings for recursions. They divided the orderings into two classes: *fair orderings*, in which no rule is applied more often than others, and *nonfair orderings*, in which some rules are applied more frequently than others. They concluded that, in the absence of information about the contents of extensional database relations, one type of fair orderings, called *cycle-preserving fair orderings*, is preferable. For example, the new order $O_2$ is a cycle-preserving fair ordering for $\mathscr{P}_1$. They also proposed the generalized semi-naive evaluation (GSN) algorithm that is based on immediate utilization of tuples and that can handle a large class of orderings including cycle-preserving fair orderings. Note that, however, there are recursions that have no cycle-preserving fair ordering [20]. Furthermore, efficient checking of the existence of a cycle-preserving fair ordering and finding such an ordering are open problems [20].

Kuittinen et al. [16] proposed an algorithm to determine an order of rule applications, which was implemented by a nested loop structure. For a recursion, the algorithm splits it into subcomponents repeatedly and selects a topological order between subcomponents. Each subcomponent is implemented by its own loop. Their evaluation technique also satisfies the immediate utilization of tuples, but it does not satisfy the semi-naive property. Their technique performs well only on the extensional database instances having some specific structures [20].

There have been a number of research results [21, 19, 22] in finding other types of ordering to optimize the evaluation of recursions. Schmidt et al. [21] proposed a technique for ordering tuples of the given extensional database instance or the intermediate results. Srivastava et al. [22] developed a framework to optimize storage space by ordering tuples produced in the evaluation of recursions. Ramakrishnan et al. [19] proposed a hybrid technique between breadth-first search based bottom-up evaluation and depth-first search based top-down evaluation. According to the depth-first search order, the technique uses recursive subgoals generated in a bottom-up manner, but computes answers in a tuple-at-a-time manner.

### 2.2. *OUR APPROACH*

We refer to all the previous evaluation techniques that set the order of rule applications at compile time as *static ordering techniques*. We observe the following drawbacks of static ordering techniques.

- Static ordering techniques cannot totally avoid the meaningless rule applications during the evaluation. As we described before, the evaluation of $\mathscr{P}_1$ by GSN based on the cycle-preserving fair ordering may include a lot of meaningless rule applications, even under the assumption that all the meaningful applications produce some new tuples. Furthermore, the assumption does not hold in general. For example, in the evaluation of $\mathscr{P}_1$ by the cycle-preserving fair ordering, let us assume that the application of $r_5$ in the first iteration does not produce any tuple. Then all the remaining four applications of $r_6$, $r_3$, $r_4$, and $r_8$ are meaningless.
- As Ramakrishnan et al. [20] pointed out, since static ordering techniques consider only the syntactic relationships between recursive rules involved in the recursion, they may have extremely bad performance on some specific extensional database instances. In order to find an optimal order of applying rules in evaluating a recursion, we should consider the structure of the extensional database instance given at evaluation time as well as the syntactic relationships between recursive rules involved in the recursion. Finding the optimal order for a given extensional database instance requires examining the extensional database instance thoroughly, which usually is not practical due to excessive cost.

A major contribution of this paper is to completely avoid meaningless rule applications by determining the order of rule applications at run time dynamically. The dynamically ordered semi-naive evaluation technique (or

simply DYN) proposed in this paper maintains the set of active rules during the evaluation. At a given time, a rule is *active* if its application at that time is meaningful and *inactive*, otherwise. We will redefine the terms active and inactive more formally in Section 4. DYN selects a rule from the set of active rules as the next rule to be applied. Note that the set of active rules is changed each time after a rule application.

Now, we introduce our technique briefly using $\mathscr{P}_1$. After applying the exit rule in $\mathscr{P}_1$, we have two active rules, $r_2$ and $r_5$. We assume that $r_2$ is selected. If the application of $r_2$ produces some new tuples for *supm*2, then $r_7$ becomes newly active. In such a case, we say that the application of $r_2$ *activates* $r_7$. Rule $r_2$ becomes inactive since this application of $r_2$ consumes all the tuples that the application of the exit rule $r_1$ produced. However, $r_5$ remains active because the tuples generated by the exit rule are not used yet. Next, we assume that $r_7$ is selected and that its application produces some new tuples for *msg*. Then the application of $r_7$ activates $r_2$ and $r_5$. Rule $r_2$ becomes active again, $r_7$ becomes inactive, and $r_5$ still remains active. At this time, the set of *msg* tuples for $r_5$ may not be equal to the set of *msg* tuples for $r_2$. The former still contains the result of the exit rule, but the latter does not since the evaluation result of the exit rule already was used in the previous application of $r_2$. Note that, so far, there have been two rule applications (the exit rule and $r_7$) that activate $r_5$, but only one rule application ($r_7$) that activates $r_2$. DYN continues such selection and application until there is no active rule.

Another major contribution of this work is the development of three selection strategies that determine the next rule from the current set of active rules. The selection strategies are designed such that the total number of rule applications can be reduced. The selection strategies use the activation state of each recursive rule and the dependency relationships between active rules as the basic information. For each recursive rule, the activation state represents whether it is active or inactive and, if it is active, how many rule applications that activated it have been made. Even after the same sequence of rule applications, the set of active rules varies according to the content of the extensional database instance used in the evaluation. The activation states and the dependency relationships can easily be obtained by maintaining only two in-memory arrays. Since the selection strategies determine the next rule by considering information about the intermediate results together with the syntactic relationships between recursive rules, DYN performs well compared with the static ordering techniques and has no extremely bad performance depending on some specific extensional database instances.

DYN also satisfies the immediate utilization of tuples and the semi-naive property. DYN iterates each recursive rule individually in the same man-

ner as the algorithms proposed by Kildall [15] and Cai and Paige [6]. For the semi-naive property, we transform recursive rules into equivalent ones that satisfy the semi-naive property by using the differential notation proposed by Balbin and Ramamohanarao [1]. There is another differential notation proposed by Bancilhon [2], but it requires more joins than the former.

## 3. PRELIMINARIES

*Datalog* is a language based on function-free Horn clause logic. We assume that the reader is familiar with the standard logic terminology [11] and the notation of datalog [25].

A datalog program consists of an extensional database (EDB) and an intensional database (IDB). The EDB is the set of tuples (facts) that are assumed to be stored explicitly in the storage. The IDB is the set of rules, each of which is of the form

$$p_0 \leftarrow p_1, p_2, \ldots, p_n.$$

For the sake of simplicity, we have omitted arguments of each predicate. We call $p_0$, the left-hand side of $\leftarrow$, the *head* of the rule, and call $p_1, p_2, \ldots, p_n$, the right-hand side of $\leftarrow$, the *body* of the rule.

Given two predicates $p$ and $q$, we say that $p$ *derives* $q$ ($p \Rightarrow q$) if $q$ is the head of a rule and $p$ occurs in the body of the rule. We say that $p$ derives $q$ transitively ($p \Rightarrow^+ q$) if $p \Rightarrow q$ or there is a predicate $s$ such that $p \Rightarrow s$ and $s \Rightarrow^+ q$. A predicate $p$ is *recursive* if $p \Rightarrow^+ p$. Two predicates $p$ and $q$ are mutually recursive to each other if $p \Rightarrow^+ q$ and $q \Rightarrow^+ p$. A rule is *recursive* if there is a predicate in its body that is mutually recursive to its head.

We refer to predicates appearing in EDB facts as *extensional predicates* and predicates appearing in the heads of rules as *intensional predicates*. As most researchers do, we assume that the set of extensional predicates and the set of intensional predicates are disjoint, i.e., no extensional predicate appears in the heads of rules in IDB. The program that has predicates appearing in both EDB facts and the heads of rules can be normalized to make two predicate sets disjoint [25]. We also assume that there is no rule whose head has constants or repeated variables. Rules that have constants and repeated variables in their heads can easily be transformed into equivalent generalized ones [12] which have neither constants nor repeated variables in their heads [25, 27]. For example, the rule $r_1$ in $\mathscr{P}_1$ has a constant in its head. It can be transformed into the following rule that has

no constant in its head:

$$msg(X) \leftarrow X = 1.$$

The evaluation result of the transformed rule is a unary relation having a single tuple whose value is 1 [24]. In this paper, we consider only bottom-up evaluable rules [4] in order to guarantee the safety of evaluation. The safety means that the final and intermediate results are finite [25, 27].

In this paper, we represent a datalog program as a graph, called *the unified rule/goal graph*, which is a simplification of the rule/goal graph proposed in [24]. The rule/goal graph for a datalog program consists of rule nodes, goal nodes, and arcs. There is a goal node $p^{\alpha}$ for each predicate $p$ and a rule node $r^{\beta}$ for each rule $r$. $\alpha$ and $\beta$ are adornments that represent the binding status of variables in a rule or arguments in a predicate. There is an arc from a rule node $r^{\beta}$ to a goal node $p^{\alpha}$ if $p$ is the head of rule $r$, and there is an arc from a goal node $p^{\alpha}$ to a rule node $r^{\beta}$ if $p$ appears in the body of rule $r$. The binding information represented by adornments is very important for the optimization focused on the relevant-data-only property. Since we do not deal with this optimization, we need not specify any adornment. The rule/goal graph without specifying any adornment is referred to as the unadorned rule/goal graph [4]. The unified rule/goal graph is equivalent to the unadorned rule/goal graph except that, instead of drawing the arcs from rule nodes to goal nodes, all nodes for the rule with the same head predicate are grouped together into the goal node for the head predicate. Thus, in the unified rule/goal graph, every rule node has only incoming arcs and every goal node has only outgoing arcs. We denote each goal node by a rectangle labeled with the predicate name and each rule node by a circle labeled with the rule number. We denote an arc from a goal (or predicate) $p_i$ to a rule $r_j$ by $(p_i, j)$. We call the goal $p_i$ the *tail* of the arc and the rule $r_j$ its *head*. For example, the unified rule/goal graph for the program $\mathscr{P}_1$ is given in Figure 2. Note that there is no rule node in the goal nodes of extensional predicates.

We can easily identify the data flow during the evaluation of a datalog program by using the unified rule/goal graph. A goal node collects the tuples produced by the rule nodes in it and sends them along its outgoing arcs. A rule node produces all the tuples that can be derived using the corresponding rule and the set of tuples that are received through its incoming arcs. We call the process for producing tuples by a rule node $r_j$ the *application* of the rule $r_j$. An arc $(p_i, j)$ represents that the tuples

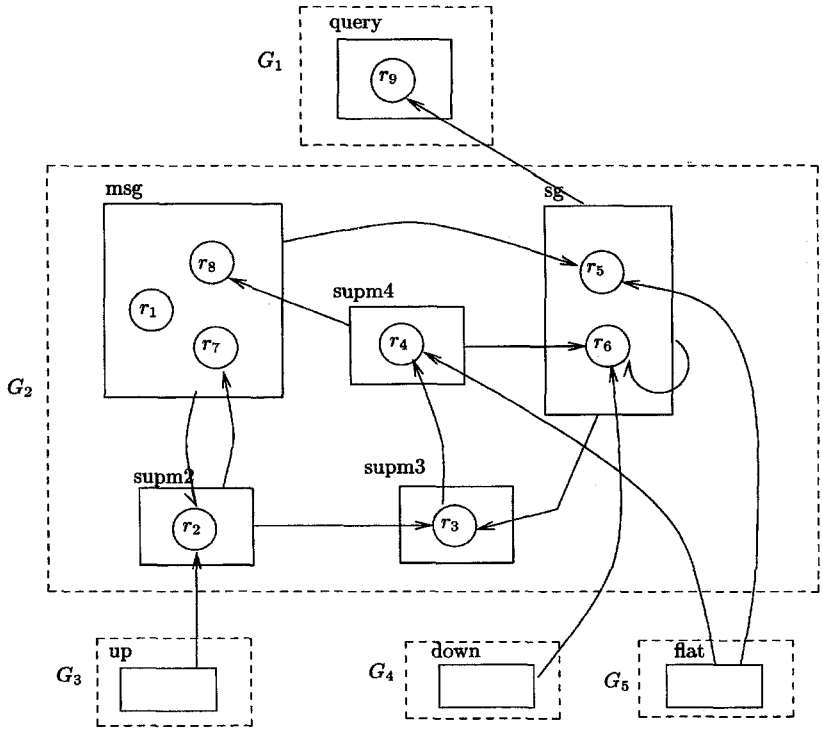



Fig. 2. The unified rule/goal graph of $\mathscr{P}_1$.

produced by applications of the rules that define the predicate $p_i$ are used in the application of the rule $r_j$.

We now define *strongly connected components* (SCC) in the unified rule/goal graph. Let the unified rule/goal graph for a datalog program be $G=(R, P, A)$, where $R$ is a set of rule nodes, $P$ is a set of goal nodes, and $A$ is a set of arcs. A strongly connected component (SCC) of $G$ is a subgraph $G'=(R', P', A')$ that satisfies the following properties:

- $P'$ is a maximum subset of $P$ such that, when we regard each arc $(p_i, j)$ as an arc from $p_i$ to the goal node of the $r_j$'s head predicate, there is a path between any pair of two goal nodes in $P'$.
- $R'$ is the set of rule nodes that belong to a goal node of $P'$.
- $A'$ is the set of arcs that connect goal nodes in $P'$ with rule nodes in $R'$.

Note that there are two types of SCCs: one is the *trivial* SCC whose set of arcs is empty; the other is the *nontrivial* SCC whose set of arcs is not empty. A trivial SCC has only one goal node. In Figure 2, there is only one nontrivial SCC, $G_2$. The arcs that are in a nontrivial SCC are referred to as *recursive arcs*, and the arcs that connect between two SCCs are referred to as *nonrecursive arcs*. For example, there are five nonrecursive arcs—(*up*, 2), (*down*, 6), (*flat*, 4), (*flat*, 5), and (*sg*, 9)—in Figure 2. All the other arcs are recursive. We define a *recursion* as a nontrivial SCC. Trivial SCCs correspond to nonrecursive predicates (or nonrecursive rules) in the datalog program. The nodes for recursive rules have at least one incoming recursive arc in the unified rule/goal graph. Note that, according to our definition of the recursion, the set of rule nodes in a recursion includes the nodes for the exit rules of the recursion. An exit rule is a nonrecursive rule whose head predicate is recursive. Rule nodes without any recursive arc in a recursion correspond to the exit rules of the recursion. In Figure 2, for example, there is only one recursion, $G_2$. $r_1$ is the exit rule of $G_2$.

## 4. THE DYNAMICALLY ORDERED SEMI-NAIVE EVALUATION

Now, we formally describe the dynamically ordered semi-naive (DYN) evaluation algorithm. Let $G_s = (R_s, P_s, A_s)$ be a recursion to be evaluated. DYN maintains a differential relation $\Delta(p_i, j)$ for each recursive arc $(p_i, j)$ in $A_s$ as well as a relation $p_i$ for each recursive predicate $p_i$ in $P_s$. The relation $p_i$ for each intensional predicate has the complete result for the predicate up to the present. At a given point, $\Delta(p_i, j)$ keeps a set of tuples that were produced by applications of some rules defining $p_i$, but haven't yet been used in any previous applications of $r_j$. For the semi-naive property, $\Delta(p_i, j)$ is set to empty after every application of the rule $r_j$, except when $r_j$ defines $p_i$. Note that if $r_j$ defines $p_i$ (in the unified rule/goal graph, the rule node for $r_j$ belongs to the goal node for $p_i$ and there is an arc from $p_i$ to $r_j$), then the application result of $r_j$ should be added to $\Delta(p_i, j)$.

We now redefine the terms *active* and *inactive* for the recursive rules on the unified rule/goal graph. We say that a recursive arc is *active* if its differential relation is not empty, and is *inactive*, otherwise. At a given point, a recursive rule is *active* if the corresponding rule node has at least one active incoming arc, and is *inactive*, otherwise. DYN maintains a set $V$ of active rules, and $V$ is initially set to empty.

DYN begins with applying all exit rules. Let $r_{e_k}$ be an exit rule defining $p_k$, i.e., the rule node $r_{e_k}$ belongs to the goal node $p_k$. The application of

$r_{e_k}$ includes the transformation of the rule to a relational algebra expression of the form $\pi_L(E)$ [25], where $L$ is the list of arguments of $p_k$ and $E$ is the relational algebra expression corresponding to the body of $r_{e_k}$. We refer to the transformed relational algebra expressions for $r_{e_k}$ as $RAE(r_{e_k})$. Note that, for the exit rules, we do not necessarily need to use the differential notation for the semi-naive property. Let $\delta r_{e_k}$ be the application result of the rule $r_{e_k}$. $\delta r_{e_k}$ is added to the relation for $p_k$ and the differential relations for all the outgoing arcs of $p_k$.

After applying all the exit rules, we have some active arcs and some active rules. Among these active rules, DYN selects a rule using some selection strategies and applies it through Algorithm EvalRNode. We will describe the details of Algorithm EvalRNode later. The application of an active rule may produce some new tuples for its head predicate. These new tuples are also added into the differential relations for all outgoing arcs of the goal node, and then the set of active rules may be changed. DYN continues this selection and application until there is no active rule, i.e., $V=\varnothing$. In the next section, we describe the selection strategies developed so that the total number of rule applications can be reduced. The dynamically ordered semi-naive evaluation algorithm for $G_s$ is given in Figure 3.

In order to explain Algorithm EvalRNode for the application of an active rule, we use the following rule $r_6$ in $\mathscr{P}_1$:

$$r_6\colon sg(X,Y) \leftarrow supm4(X,Z), sg(Z,W), down(Z,Y).$$

Here, *sg* and *supm*4 are mutually recursive to the head, and *down* is an EDB predicate. Figure 4 shows the part of the unified rule/goal graph around the rule $r_6$. If $r_6$ is active, then $\Delta(supm4,6)$, $\Delta(msg,6)$, or both are not empty. The following expression is the version of $r_6$ resulting from the semi-naive rewriting based on the differential notation proposed in [1] (for the sake of simplicity, we have omitted the join conditions and the projection list):

$$\pi\left(\Delta(supm4,6) \bowtie sg \bowtie down \cup supm4^{\text{old}} \bowtie \Delta(sg,6) \bowtie down\right) - sg.$$

Here, $supm4^{\text{old}} = supm4 - \Delta(sump4,6)$. For an arbitrary recursive rule $r_j$, we refer to a semi-naive rewritten version of $r_j$ as $semiRAE(r_j)$ and the application result of $r_j$ at a time as $\delta r_j$. Then, we have $\delta r_6 = semiRAE(r_6)$. Note that $\delta r_j$ is the application result of $r_j$, i.e., the set of new tuples that are generated by an application of $r_j$, while the differential relation $\Delta(p_i,j)$ maintains all the $p_i$ tuples that have not yet been used in any previous application of $r_j$.

```
Algorithm DYN(G_s)
{
    (1) for every p_i ∈ P_s, p_i = ∅;
    (2) for every recursive arc, (p_i, j) ∈ A_s, Δ(p_i, j) = ∅;
    (3) V = ∅;
    (4) for every exit rule r_{e_k}, do begin
        /* Let p_k (∈ P_s) be the head predicate of r_{e_k}. */
        (5) δr_{e_k} = RAE(r_{e_k});
        (6) for every p_k's outgoing arc (p_k, j), do begin
            (7) Δ(p_k, j) = Δ(p_k, j) ∪ δr_{e_k};
            (8) V = V ∪ {r_j};
        (9) endfor;
        (10) p_k = p_k ∪ δr_{e_k};
    (11) endfor;
    (12) while ( V ≠ ∅) do begin
        (13) select a rule, r_j, from V using some selection strategies
        (14) EvalRNode(r_j);
    (15) endwhile;
}
```

Fig. 3. Algorithm DYN.

After computing the above equation, to ensure the semi-naive property, we set the differential relations for all the incoming arcs of $r_6$ to be empty; i.e., $\Delta(supm4,6) = \Delta(sg,6) = \varnothing$. Then we add $\delta r_6$ into $sg$ and the differential relations for all outgoing arcs of $sg$, $\Delta(sg,6)$ and $\Delta(sg,3)$.

The general description of Algorithm EvalRNode is given in Figure 5. Note that by lines (4) and (8) of Algorithm EvalRNode, the set $V$ of active rules may vary after each application of an active rule.

DYN can also be used with a static order by modifying the loop contents. Instead of line (13) and (14), we call Algorithm EvalRNode for

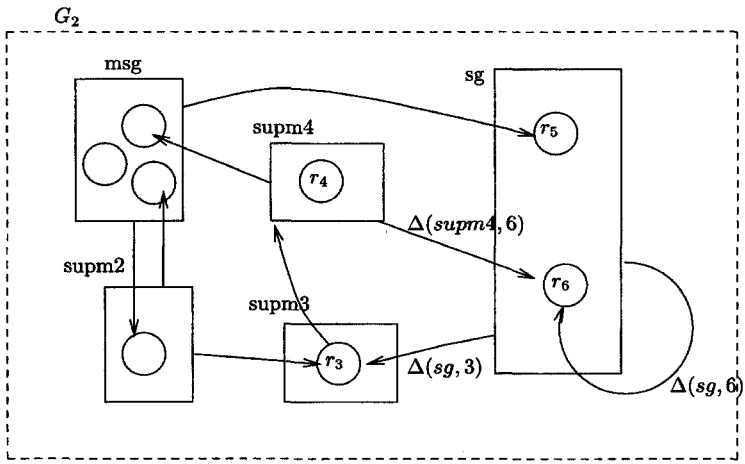



Fig. 4. The part of the unified rule/goal graph in Figure 2.

each recursive rule in the static order. The resulting algorithm from this modification of DYN is similar to GSN [20]. However, for each recursive arc $(p_i, j)$ while GSN maintains the old version of the relation $p_i$ with respect to the rule $r_j$, DYN maintains the differential relation $\Delta(p_i, j)$. In general, the differential relation is much smaller than the old version of the relation. Therefore, DYN requires a relatively small amount of space for the intermediate data.

## 5. SELECTION STRATEGIES

In general, we may have more than one active rule at a time during the evaluation of a recursion. In this section, we describe three selection strategies for determining the next rule to be applied among these active rules. The goal of these strategies is to minimize the total number of rule applications by maximizing the effect of each rule application in the evaluation.

First, we define some notation useful for explaining selection strategies. If there is a recursive arc $(p, j)$ in a recursion, then we say that the rule $r_j$ is *directly dependent* on each rule defining the predicate $p$. The direct dependencies between rules represent the direction of data flow. For example, in the recursion shown in Figure 6, the rule $r_6$ is directly dependent on the rule $r_4$, but the rule $r_4$ is not directly dependent on the rule $r_6$. The fact that the rule $r_6$ is directly dependent on the rule $r_4$ means

Algorithm EvalRNode($r_j$)

/* Let $r_j$ be $p_0 \leftarrow p_1, p_2, \ldots, p_n, q_1, \ldots, q_m$,

where $p_i$'s, $1 \leq i \leq n$, are mutually recursive to $p_0$ and $q_k$'s, $1 \leq k \leq m$, are not. */

{

(1) if $n > 1$, then

for each $i$, $2 \leq i \leq n, p_i^{old} = p_i - \Delta(p_i, j)$;

(2) $\delta r_j$ =semiRAE($r_j$);

(3) for each $i$, $1 \leq i \leq n, \Delta(p_i, j) = \emptyset$;

(4) $V = V - \{r_j\}$;

(5) if $\delta r_j \neq \emptyset$ then begin

(6) for every outgoing arc $(p_0, l)$ of $p_0$, do begin

(7) $\Delta(p_0, l) = \Delta(p_0, l) \cup \delta r_j$;

(8) $V = V \cup \{r_l\}$;

(9) endfor;

(10) $p_0 = p_0 \cup \delta r_j$;

(11) endif;

}

Fig. 5. Algorithm EvalRNode.

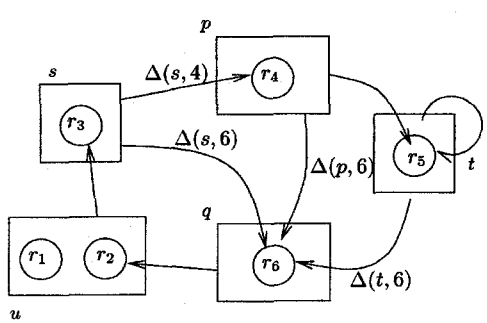



Fig. 6. An example recursion.

that if an application of rule $r_4$ produces some new tuples, then we must apply the rule $r_6$ subsequently by using these new tuples. $\Delta(p,j)^i$ denotes $\Delta(p,j)$ after the $i$th rule application. Note that $\Delta(p,j)^i$ and $\Delta(p,j)^k$ may be different when $i \neq k$.

The first selection strategy is as follows.

SELECTION STRATEGY 1 (S1). If we have more than one active rule, then we select the one that is not directly dependent on other active rules including itself.

This selection strategy is based on the following observation. Let us assume that after $i$ rule applications, there are only two active arcs $\Delta(s,4)^i$ and $\Delta(s,6)^i$ in the recursion shown in Figure 6. Then, we have two active rules, $r_4$ and $r_6$. This means that each of them should be applied at least once before the end of the evaluation. If we apply $r_6$ with $\Delta(s,6)^i$ first, then we may need another application of $r_6$ after the application of $r_4$ with $\Delta(s,4)^i$. The reason is that, if the later application of $r_4$ using $\Delta(s,4)^i$ produces some new tuples, then $r_6$ becomes active again. However, if we apply $r_4$ with $\Delta(s,4)^i$ first, then the new tuples produced by this application will be used with $\Delta(s,6)^i$ in the future application of $r_6$. Thus, if we apply an active rule that is directly dependent on other active rules, we may need additional applications of it after the applications of those other active rules.

Consecutive applications of a particular rule may have an adverse effect on maximizing the effect of each rule application while there are some other active rules. The rule that is dependent on itself may be applied consecutively until its application does not produce any new tuple, if we do not consider the self-dependency in the strategy S1. In Figure 6, let us assume that only two rules, $r_5$ and $r_6$, are active after a number of rule applications. Rule $r_5$ is not directly dependent on the other active rule $r_6$, but is directly dependent on itself. If we do not consider the self-dependency, then $r_5$ is selected for the next application. Since $r_5$ remains active as long as its application produces some new tuples, the other active rule $r_6$ cannot be selected while the application of $r_5$ produces some new tuples.

Before we describe the second selection strategy, we define the *activation rate* $AR_{r_j}$ of an active rule $r_j$ as

$$AR_{r_j} = ACT_{r_j} / I_{r_j}.$$

Here, $I_{r_j}$ is the number of incoming recursive arcs of $r_j$, i.e., the number of recursive predicates in its body. $ACT_{r_j}$ is the number of rule applications that activated $r_j$. $ACT_{r_j}$ is set to zero each time after applying $r_j$, and is

incremented by 1 if some new tuples are produced by applying a rule on which $r_j$ is directly dependent. Consider the following sequence $s1$ of rule applications for the recursion in Figure 6:

$$s1: \ldots, r_4^i, r_6^{i+1}, r_2^{i+2}, r_3^{i+3}, r_4^{i+4}, r_5^{i+5}, r_6^{i+6}, \ldots.$$

In this notation of the rule application sequence, $r_4^i$ means the $i$th rule is $r_4$. If all the applications in the sequence produce some new tuples, then the activation rate $AR_{r_6}$ after the $(i+5)$th application is 1 $(=3/3)$, i.e., $I_{r_6}=3$ and $ACT_{r_6}=3$. At the point after the $(i+5)$th application, there have been four applications including the $(i+5)$th application since the recent [$(i+1)$th] application of $r_6$. However, since $r_6$ is not directly dependent on $r_2$, there have been three applications that activated $r_6$. If the $(i+4)$th application does not produce any tuple, then $AR_{r_6}$ is $2/3$ at that point.

When we consider each rule individually, it is better to delay its application as late as possible. The delayed application makes more inferences than the early application. For example, consider another sequence $s2$ of rule applications for the recursion in Figure 6:

$$s2: \ldots, r_4^i, r_6^{i+1}, r_2^{i+2}, r_3^{i+3}, r_6^{i+4}, \ldots.$$

Let us assume that $s1$ and $s2$ have the same subsequence until the $(i+1)$th application. After the $(i+1)$th application, $s2$ applies $r_6$ faster than $s1$. Let us assume that all the rule applications in both $s1$ and $s2$ produce some new tuples. The $(i+6)$th application of $r_6$ in $s1$ uses all the results of three rule applications that activated $r_6$, while the $(i+4)$th application of $r_6$ in $s2$ uses only the result of the $(i+3)$th application. It is obvious that the delayed application of $r_6$ in $s1$ makes more (at least the same) inferences than the early application in $s2$, because the former uses larger (at least the same) number of tuples than the latter. When we consider all the active rules together, we expect that the one having the maximum activation rate may make relatively more inferences than others.

SELECTION STRATEGY 2 (S2). We choose the rule having the maximum activation rate as the next rule.

The main objective of selection strategy 2 is to avoid repeated applications of only those rules that form a particular cycle, even if there are some other active rules in other cycles. The following example shows the possibility of repeated application of rules in a particular cycle of a recursion.

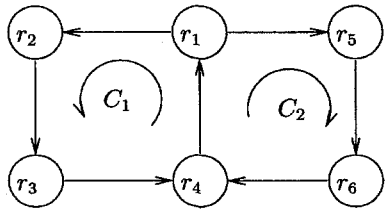



Fig. 7. A recursion with two cycles.

EXAMPLE 1. Consider a recursion shown in Figure 7, which consists of two cycles $C_1$ and $C_2$. Each arc in Figure 7 represents that the rule of its head is directly dependent on the rule of its tail. For the sake of simplicity, we assume that each rule defines a distinct predicate. A cycle $C_1$ consists of four rules, $r_1$, $r_2$, $r_3$, and $r_4$, and the other cycle, $C_2$, consists of $r_1$, $r_5$, $r_6$, and $r_4$. Assume that the rule $r_1$ is the only active rule after the $i$th application. If the $(i+1)$th application of $r_1$ produces some new tuples, then two rules, $r_2$ and $r_5$, become active. Since these two rules are not directly dependent on each other, we cannot select one for the next application by S1. If we choose one arbitrarily, then we may have the sequence of rule applications shown in the second column (Only S1) of Table 1. Thus, only the four rules that are involved in $C_1$ are applied repeatedly until an application of one of them produces no new tuple, even if there is an active rule $r_5$ in the other cycle, $C_2$.

For each cycle $C$ in a recursion, we refer to the rule nodes that are directly dependent on some nodes in $C$ but belong to other cycles as the

TABLE 1
Two Possible Sequences of Rule Applications

| Rule application no. | Only S1 | | S1 and S2 | |
|---|---|---|---|---|
| | Selected rule | Active rules after applying the selected rule | Selected rule | Active rules after applying the selected rule |
| $i+1$ | $r_1$ | $\{r_2, r_5\}$ | $r_1$ | $\{r_2(1.0), r_5(1.0)\}$ |
| $i+2$ | $r_2$ | $\{r_3, r_5\}$ | $r_2$ | $\{r_3(1.0), r_5(1.0)\}$ |
| $i+3$ | $r_3$ | $\{r_4, r_5\}$ | $r_3$ | $\{r_4(0.5), r_5(1.0)\}$ |
| $i+4$ | $r_4$ | $\{r_1, r_5\}$ | $r_5$ | $\{r_4(0.5), r_6(1.0)\}$ |
| $i+5$ | $r_1$ | $\{r_2, r_5\}$ | $r_6$ | $\{r_4(1.0)\}$ |
| $i+6$ | ⋮ | ⋮ | $r_4$ | $\{r_1(1.0)\}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

*neighbor nodes* of $C$. Since a recursion is a strongly connected component, $C$ has at least one neighbor node, and at least one rule node in $C$ is a neighbor node of another cycle if the recursion has two or more cycles and at least one cycle includes two or more rule nodes. Among all the neighbor nodes of $C$, we call the one having minimum incoming arcs the *next-turn neighbor node* of $C$. Let $k$ be the number of incoming arcs of the next-turn neighbor node of $C$. The activation rate of the next-turn neighbor node is incremented by at least $1/k$ each time after applying all the rules in $C$. Therefore, after $k$ consecutive repetitions of $C$, the activation rate of the next-turn neighbor node becomes larger than or equal to 1. In the meanwhile, the nodes in $C$ that are neighbor nodes of other cycles have two or more incoming arcs: one from another node in $C$ and all the others from nodes in other cycles. This means that the activation rates of those nodes in $C$ are less than 1 as long as only $C$ is repeated consecutively. Therefore, by S2, $C$ cannot be repeated more than $k$ times consecutively. As the result of the discussion so far, we conclude that, using S2, the maximum number of consecutive repetitions of a particular cycle is limited to the number of incoming arcs of its next-turn neighbor node.

For example, the next-turn neighbor node of the cycle $C_1$ in Figure 1 is $r_5$. Using S2, there is no consecutive repetition $C_1$, since the number of incoming arcs of $r_5$ is 1. The last column (S1 and S2) of Table 1 gives the sequence of rule applications for the recursion of Example 1 by S2. We also show the activation rate for each active rule. After the $(i+3)$th rule application, we have two active rules, $r_4$ and $r_5$. The activation rate of $r_4$ (0.5) is less than that of $r_5$ (1.0). Then, by S2, $r_5$ is selected as the rule for the $(i+4)$th application.

Unfortunately, using S2 with S1, we do not completely avoid repetitive rule applications along a particular cycle for every possible recursion. The recursion shown in Figure 8 is an example whose evaluation has such repetitive rule applications even if we use S2 in the rule selection. Two rules, $r_i$ and $r_j$, are directly dependent on each other, and both of the two rules are not directly dependent on themselves. Besides, there is another rule $r_k$, which is directly dependent on both of these two rules. Suppose an application of $r_i$ produces some new tuples. Then, we have two active rules, $r_j$ and $r_k$. By S1, we select $r_j$ as the next rule to be applied. If the application of $r_j$ also produces some new tuples, then $r_i$ becomes active again. At that time, even if the activation rate of $r_k$ is larger than that of $r_i$, we select $r_i$ again since S1 has a higher priority over S2. For such a recursion, we prevent the repetition of rule applications along a particular cycle by assigning a threshold value for the activation rate. Thus, if the activation rate of an active rule is greater than the threshold value, then we select it without considering S1.

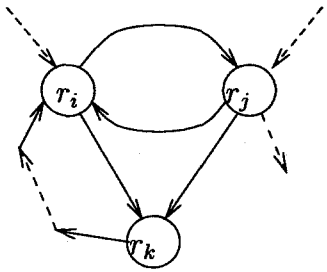



Fig. 8. An exceptional case of S1 and S2.

We can view the selection of the next rule by S1 and S2 as a priority-based scheduling process. S1 and S2 define the priority of a rule $r_i$ in a recursion as

$$S(r_i) \times \left(T(r_i) \times c + AR_{r_i}\right).$$

Here, $S(r_i)$ is a function whose value is 1, if $r_i$ is active, and 0, otherwise. $T(r_i)$ is also a function whose value is 1, if $r_i$ is not directly dependent on some other active rules, and 0, otherwise. $c$ is a threshold value for the activation rate. For a recursion with $n$ recursive rules and $s$ cycles, we use $n \times s$ as the threshold value in this paper. $n \times s$ is a simple estimation of the maximum number of rule applications after all the $s$ cycles are evaluated, since each cycle may have maximum $n$ rules and a rule may belong to every cycle. We select the rule with the highest priority as the next rule to be applied. Note that the priority of each rule is changed dynamically after a rule application, and the priority of an inactive rule is zero.

When more than one active rule remains after applying the above two selection strategies, we select one among them by the following selection strategy.

SELECTION STRATEGY 3 (S3). Select the active rule with the smallest number of joins expected in its application.

It is ideal to select the active rule with the lowest cost expected in its application. However, how to estimate the cost of a rule application is beyond the scope of this paper. In order to estimate the cost, we may have to consider various factors such as the size of EDB relations, the size of each differential relation at the application time, the join selectivities between body predicates of the rule, the method of joins, and so on. In this

paper, we use the number of joins required in a rule application as a simple estimation of its cost.

The previous studies [20, 16] assumed that the number of joins was proportional to the number of rule applications. However, it is not always true. Consider a recursion with two recursive rules, $r_1$ and $r_2$. The rules $r_1$ and $r_2$ require one and three joins per application, respectively. Suppose there are two different evaluation techniques, called $A$ and $B$. For a given EDB instance, suppose that $A$ applies $r_1$ seven times and $r_2$ five times. Then $A$ performs a total of 12 rule applications and a total of 22 joins. Suppose that $B$ applies $r_1$ three times and $r_2$ seven times for the same EDB. Then $B$ performs a total of 10 rule applications and a total of 24 joins. In total, $A$ performs two rule applications more than $B$ does, but $A$ performs two joins less than $B$ does. Thus, even if a technique is better than the other technique in terms of the total number of rule applications, the former may be worse than the latter in terms of the number of joins when the former applies the rules that require a large number of joins more than the latter does.

The number of joins required in a rule application is calculated as follows. Consider an active rule $r_j$ of the form

$$r_j\colon p_0 \leftarrow p_1, \ldots, p_n, q_1, \ldots, q_m.$$

Here, $p_i$'s, $1 \le i \le n$, are mutually recursive with $p_0$, and $q_i$'s, $1 \leqslant i \leqslant m$, are not. The semi-naive version of $r_j$,

$$\begin{aligned}
\delta r_j = (&\Delta(p_1, j) \bowtie \quad p_2 \bowtie p_3 \bowtie \cdots \quad \bowtie \quad p_n \bowtie q_1 \bowtie \cdots \bowtie q_m \\
\cup\ & p_1^{\text{old}} \bowtie \Delta(p_2, j) \bowtie p_3 \bowtie \cdots \quad \bowtie \quad p_n \bowtie q_1 \bowtie \cdots \bowtie q_m \\
\cup\ & \qquad\qquad\qquad \cdots \\
\cup\ & p_1^{\text{old}} \bowtie \quad p_2^{\text{old}} \bowtie \quad \cdots \bowtie p_{n-1}^{\text{old}} \bowtie \Delta(p_n, j) \bowtie q_1 \bowtie \cdots \bowtie q_m\ ) \\
& - p_0,
\end{aligned}$$

involves the union of $n$ join terms. There is one join term for each occurrence of recursive body predicates.

We assume that each pair of adjacent body predicates needs only one join operation. Then there are $n+m-1$ join operations in each join term. We have to calculate all the join terms with the nonempty differential relations. Therefore, if there are $k$ $(1 \leqslant k \leqslant n)$ active arcs, then the number of join operations required in an application of $r_j$ is $k \times (n+m-1)$.

## 6. EXPERIMENTAL RESULTS

In this section, we present a performance study on the effect of dynamic ordering of rule applications. We compare our technique (DYN) with the following three different techniques through the similar way used in the performance study of [20].

- BSN: The basic semi-naive evaluation technique in which the order of rule applications is not specified.[1]
- GSN: The generalized semi-naive evaluation technique proposed by [20] in which the order of rule applications is a cycle-preserving fair ordering of a given recursion.
- NESTED: The evaluation technique proposed by [16] in which rules are applied in a nested loop.

We use the same two datalog programs, $\mathscr{P}_1$ and $\mathscr{P}_2$, as used in [20]. $\mathscr{P}_1$ has already been given in Figure 1. $\mathscr{P}_2$ (see Appendix) is a version of another nonlinear same generation program rewritten by using the counting technique [5]. For each evaluation technique, we measure the total number of rule applications and the total number of joins.

In order to show the benefit of DYN, we first compare our technique with BSN and GSN by using $\mathscr{P}_1$. For GSN, we use the cycle-preserving fair ordering $(r_2, r_7, r_5, r_6, r_3, r_4, r_8)$. The EDB instances we use for $\mathscr{P}_1$ are $A_n$, $B_n$, $C_n$, $F_n$, $T_{n,m}$, and $U_{n,m}$. $A_n$ and $B_n$ are the same data sets as used in [16], and $C_n$ and $F_n$ are the same data sets as used in [20]. $T_{n,m}$ and $U_{n,m}$ are from [4]. The graphical description of these data sets is presented in the Appendix. $A_n$ has a unique path from a node to its same generation, i.e., there is a unique sequence of inferences to get the answer, but it requires a large number of rule applications. $B_n$ and $C_n$ have a moderate number of different paths between two nodes in the same generation and takes a moderate number of rule applications. $C_n$ takes less rule applications than $B_n$, but has more paths than $B_n$. $F_n$, $T_{n,m}$, and $U_{n,m}$ have a large number of different paths between two nodes in the same generation, but require a small number of rule applications.

Then we compare our technique with GSN and NESTED by using $\mathscr{P}_2$. The EDB instances we use for $\mathscr{P}_2$ are $C_n$ and $S_n$, which were used in [20]. $C_n$ is designed in such a way that NESTED performs well, and $S_n$ is designed in such a way that NESTED performs very badly compared to GSN [20]. The graphical description of these two data sets and the static

[1] In the implementation, we use the order given by the user, i.e., the user written order.

TABLE 2
Total Number of Rule Applications: $\mathscr{P}_1$

| Data sets | BSN | GSN | DYN | GSN-m |
|---|---|---|---|---|
| $A_4$ | 357 | 105 | 67 | 70 |
| $A_{10}$ | 25,053 | 7,161 | 4,647 | 4,984 |
| $B_4$ | 203 | 63 | 39 | 48 |
| $B_{64}$ | 1,029 | 476 | 189 | 466 |
| $C_4$ | 161 | 49 | 36 | 42 |
| $C_{16}$ | 329 | 126 | 66 | 115 |
| $F_5$ | 91 | 35 | 20 | 26 |
| $F_{10}$ | 161 | 49 | 40 | 39 |
| $F_{15}$ | 231 | 140 | 147 | 127 |
| $T_{3,3}$ | 112 | 42 | 29 | 31 |
| $T_{5,3}$ | 161 | 70 | 48 | 59 |
| $U_{3,3}$ | 77 | 35 | 22 | 25 |
| $U_{5,10}$ | 154 | 56 | 36 | 43 |

orderings for GSN and NESTED are also presented in the Appendix. Through this performance comparison, we try to show that, while the performance of static ordering techniques may vary depending on EDB instances, our technique performs steadily well on every possible EDB instance.

Table 2 shows the total number of rule applications performed by each evaluation technique on $\mathscr{P}_1$. The column GSN-m gives the total number of only the meaningful rule applications taken by GSN. DYN is better than two other evaluation techniques for every data set we considered. DYN performs about 70% ($T_{5,3}$) to 81% ($F_5$) better than BSN, and performs about 18% ($F_{10}$) to 60% ($B_{64}$) better than GSN.

The great reduction in the total number of rule applications resulted from the fact that the rule applications taken by DYN are all meaningful. As we mentioned in Section 2, the static ordering techniques may take a lot of meaningless rule applications. We can get the number of meaningless rule applications in GSN by subtracting the value of column GSN-m from the value of column GSN.

DYN is also better than GSN-m in most cases.[2] This means that the sequence of rule applications done by DYN is better than that by GSN. Thus, the dynamic ordering determined by the selection strategies is better than the cycle-preserving fair ordering that was proved as the best among the static orderings in [20]. Note that for data sets on which GSN performs

[2] For the exceptional cases $F_{10}$ and $F_{15}$, we will give an analysis later.

relatively less number of meaningless rule applications, DYN is much better than GSN (for example, see $B_{64}$ and $C_{16}$).

Table 3 shows the total number of joins taken by each evaluation technique on $\mathscr{P}_1$. There are two columns for each technique. NoRJ gives the number of real (nonnull) joins and NoNJ gives the number of null joins. The null join is a join in which one of its operand relations is empty, and the result of such a join is empty. In the meaningful rule application, we may have null joins if the rule has more than one recursive predicate in its body. Consider a rule having two recursive body predicates. As we mentioned in the previous section, the semi-naive relational algebra equation has two join terms: one includes the differential relation for one of the recursive body predicates, and the other includes that for the remaining recursive body predicate. The rule becomes active even when either of these two differential relations is not empty.

Table 3 shows that DYN need fewer joins than the other two evaluation techniques on all the data sets we consider. For the nonnull joins, DYN is about 52% ($U_{3,3}$) to 88% ($B_{64}$) better than BSN and is about 3% ($A_4$) to 75% ($B_{64}$) better than GSN. For a data set $F_{10}$, DYN requires one more nonnull join than GSN does. However, on the total number of joins (NoRJ + NoNJ), DYN is also better than GSN for that data set.

The reduction in the total number of real joins and rule applications shown in Tables 2 and 3 is the strong evidence of that our selection strategies are good in maximizing the effect of a rule application (or a

TABLE 3

Total Number of Joins: $\mathscr{P}_1$

| | BSN | | GSN | | DYN | |
|---|---|---|---|---|---|---|
| Data sets | NoRJ | NoNJ | NoRJ | NoNJ | NoRJ | NoNJ |
| $A_4$ | 192 | 216 | 70 | 50 | 68 | 16 |
| $A_{10}$ | 15,307 | 13,325 | 5,612 | 2,572 | 4,901 | 773 |
| $B_4$ | 143 | 89 | 52 | 20 | 40 | 7 |
| $B_{64}$ | 1,104 | 72 | 530 | 14 | 130 | 7 |
| $C_4$ | 122 | 62 | 44 | 12 | 36 | 6 |
| $C_{16}$ | 299 | 77 | 129 | 15 | 54 | 6 |
| $F_5$ | 46 | 58 | 24 | 16 | 17 | 4 |
| $F_{10}$ | 104 | 80 | 42 | 14 | 43 | 4 |
| $F_{15}$ | 167 | 97 | 160 | 16 | 149 | 5 |
| $T_{3,3}$ | 71 | 57 | 33 | 15 | 29 | 4 |
| $T_{5,3}$ | 138 | 46 | 65 | 15 | 48 | 4 |
| $U_{3,3}$ | 44 | 44 | 26 | 14 | 21 | 4 |
| $U_{5,10}$ | 126 | 50 | 47 | 17 | 34 | 3 |

TABLE 4
Total Number of Rule Applications: $\mathscr{P}_2$

| Data sets | GSN | NESTED | DYN | GSN-m |
|---|---|---|---|---|
| $C_{16}$ | 204 | 174 | 124 | 148 |
| $S_{64}$ | 580 | 2,533 | 573 | 573 |

join). Remember that three techniques are all equivalent in the total number of inferences made for a given data set, since they all satisfy the semi-naive property, and any semi-naive evaluation techniques do not repeat the same inference [13]. Therefore, DYN makes more inferences per rule application (or join) than the other techniques on average. The number of null joins is also reduced by DYN significantly because DYN applies only active rules.

$F_{10}$ and $F_{15}$ are actual examples of our argument that the number of joins is not always proportional to the number of rule applications. For those data sets, GSN performs less rule applications than DYN does, but GSN requires more joins than DYN.

Table 4 gives the total number of rule applications performed by each evaluation technique on $\mathscr{P}_2$. We can see that NESTED is better than GSN on $C_{16}$, but, for $S_{64}$, it is much worse than the other two techniques. For $C_{16}$, DYN needs about 30% less rule applications than NESTED and about 40% less rule applications than GSN. Even without considering meaningless rule applications in GSN, DYN is about 16% better than GSN on $C_{16}$ in terms of the total number of rule applications. For $S_{64}$, the total number of rule applications taken by DYN is equal to that of GSN-m. In fact, both of these techniques perform the same sequence of meaningful rule applications for this data set.

Table 5 shows the total number of joins taken by each evaluation technique on $\mathscr{P}_2$. For $C_{16}$, DYN takes less number of nonnull joins than both DSN and NESTED. The comparison results in Tables 4 and 5 indicate the appropriateness of the information (the activation state and

TABLE 5
Total number of joins: $\mathscr{P}_2$

| | GSN | | NESTED | | DYN | |
|---|---|---|---|---|---|---|
| Data sets | NoRJ | NoNJ | NoRJ | NoNJ | NoRJ | NoNJ |
| $C_{16}$ | 392 | 168 | 307 | 126 | 255 | 52 |
| $S_{64}$ | 1,391 | 88 | 5,297 | 2,041 | 1,391 | 77 |

the dependency relationships between active rules) for excluding the extremely bad performance depending on the content of some specific extensional database instances. Therefore, DYN based on the selection strategies using this information shows steadily good performance.

## 7. CONCLUSION

In general, bottom-up fixed point evaluation techniques evaluate a recursion by applying recursive rules of the recursion repeatedly until no new tuples are generated, using a given extensional database instance. The order in which the rules are applied is not specified in conventional fixed point evaluation techniques. However, we can speed up the evaluation by applying rules in an appropriate order [20, 16].

We have proposed a new fixed point evaluation technique, called the *dynamically ordered semi-naive evaluation* (or simply DYN). DYN consists of a semi-naive algorithm and a set of selection strategies. The semi-naive algorithm allows dynamic ordering of rule applications and makes tuples generated by a rule application immediately available in the subsequent rule applications. After each rule application, the selection strategies determine the next rule by considering the syntactic structure of recursion and some information about the intermediate result up to the present. We have developed the selection strategies so that the total number of rule applications can be minimized by maximizing the effect of each rule application. The effect of a rule application is defined as the number of inferences made by the rule application. Since any fixed point evaluation algorithms that satisfy the semi-naive property do not repeat the same inference, they are equivalent in the total number of inferences made during the evaluation of a recursion for a given extensional database instance. Therefore, making each rule application produce more inferences can reduce the total number of rule applications. The fact that more inferences are made by each rule application implies that more inferences can be made using a set of tuples in a page fetched from disk. Thus, we can expect that the total number of I/O operations is reduced [20].

There have been some research results [20, 16] for finding a good order for rule applications. Since all the previous results fix the order of rule applications at compile time, we call them *static ordering* techniques. To find an order of a given recursion, these static ordering techniques consider only the syntactic structure of the recursion. The performance of static ordering techniques may vary according to the extensional database instance given at evaluation time, because the optimal order for the recursion also depends on the content of the extensional database in-

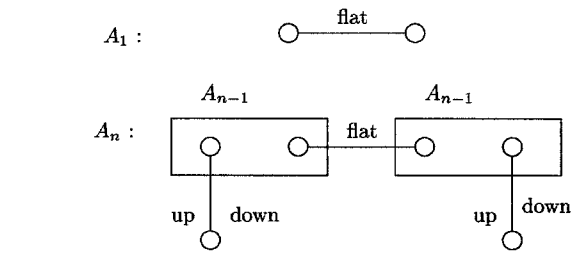
stance. Since the selection strategies also consider some information on the intermediate result, which varies according to the extensional database instance, when determining the next rule to by applied, DYN performs well compared with the static ordering techniques, not depending on the content structures of some specific extensional database instances. Through experimental comparisons, we have shown that DYN outperforms the static ordering techniques in terms of the total number of rule applications and joins.

## APPENDIX

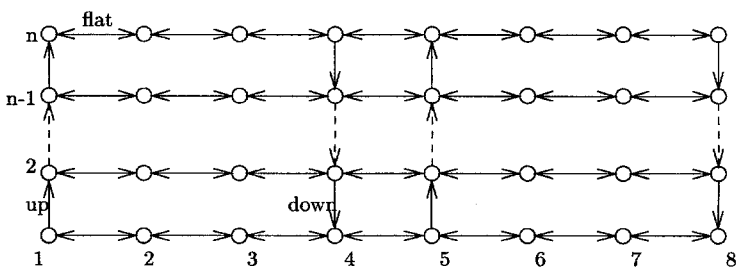
$\mathscr{P}_2$ is given in Figure 9. In $\mathscr{P}_2$, there are two recursions: one consists of four recursive rules—$r_1$, $r_2$, $r_7$ and $r_8$—and the other five recursive rules —$r_3$, $r_4$, $r_6$, $r_9$, and $r_{10}$. The cycle-preserving fair ordering for the former recursion used in GSN is $(r_7, r_1, r_2, r_8)$, and that for the latter is $(r_9, r_{10}, r_3, r_4, r_6)$. The nested ordering for the former recursions is

$$
\begin{array}{lll}
(\mathcal{P}_2) & r_1: & anc(X,Y,1) \leftarrow manc(X), up(X,Y).\\
& r_2: & anc(X,Y,N) \leftarrow N>1, manc(X), anc(X,Z,N-1), up(Z,Y).\\
& r_3: & desc(X,Y,1) \leftarrow mdesc(X,1), down(X,Y).\\
& r_4: & desc(X,Y,N) \leftarrow N>1, mdesc(X,N), desc(X,Z,N-1), down(Z,Y).\\
& r_5: & sg(X,Y) \leftarrow msg(X), flat(X,Y).\\
& r_6: & sg(X,Y) \leftarrow msg(X), anc(X,X_1,N), flat(X_1,X_2),\\
& & \qquad sg(X_2,Y_2), flat(Y_2,Y_1), desc(Y_1,Y,N).\\
& r_7: & manc(X) \leftarrow msg(X).\\
& r_8: & msg(X_2) \leftarrow msg(X), anc(X,X_1,N), flat(X_1,X_2).\\
& r_9: & mdesc(Y_1,N) \leftarrow msg(X), anc(X,X_1,N), flat(X_1,X_2), sg(X_2,Y_2), flat(Y_2,Y_1).\\
& r_{10}: & mdesc(X,N-1) \leftarrow mdesc(X,N), N>1.\\
& r_{11}: & msg(1).\\
& r_{12}: & query(Y) \leftarrow sg(1,Y).
\end{array}
$$

Fig. 9. $\mathscr{P}_2$.




(a) $A_n$



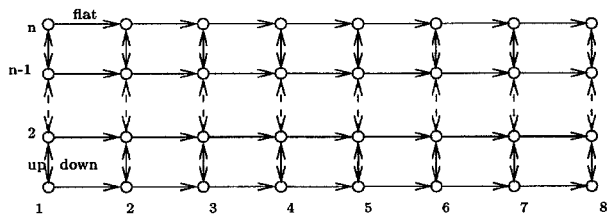

(b) $B_n$

Fig. 10. Data sets $A_n$ and $B_n$.

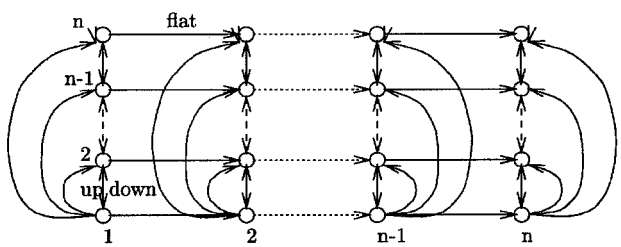
$(r_7, r_1, (r_2), r_8)$,[3] and that for the latter is $(r_9, (r_{10}), r_3, (r_4), r_6)$. All the orderings used in this paper are the same as those used in [20].

In Figure 10, we depict data sets $A_n$ and $B_n$ which are the same as those used in [16]. The left and right arrow-headed arcs represent *flat* tuples, and the up and down arrow-headed arcs represent *up* and *down* tuples, respectively. We also give the graphical descriptions of three data sets $C_n$, $F_n$, and $S_n$ in Figure 11. They are the same as those used in [20]. We take two other data sets $T_{n,m}$ and $U_{n,m}$ depicted in Figure 12. $T_{n,m}$ is a tree-structured data set, where $n$ is the height of a tree and $m$ is the fan-out of each node in the tree. $U_{n,m}$ is a cylindric data set, where $n$ is the base of a cylinder and $m$ is the height of the cylinder. $T_{n,m}$ and $U_{n,m}$ are used in [4] for comparing the performances of the well-known evaluation

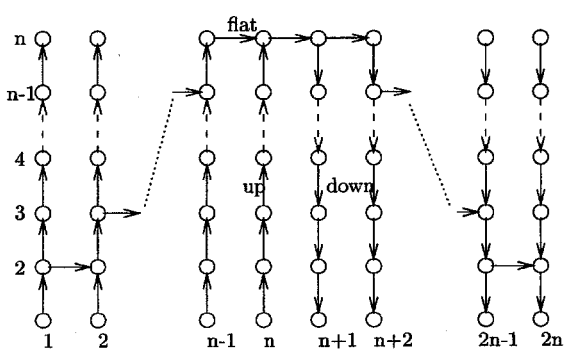
[3] In this representation of orderings, each pair of parentheses is implemented by a loop. In this case, we have two loops: one is the inner loop that applies $r_2$ repeatedly, and the other is the outer loop that applies three rules and the inner loop repeatedly by the given order.




(a) $C_n$




(b) $F_n$




(c) $S_n$

Fig. 11. Data sets $C_n$, $F_n$, and $S_n$.

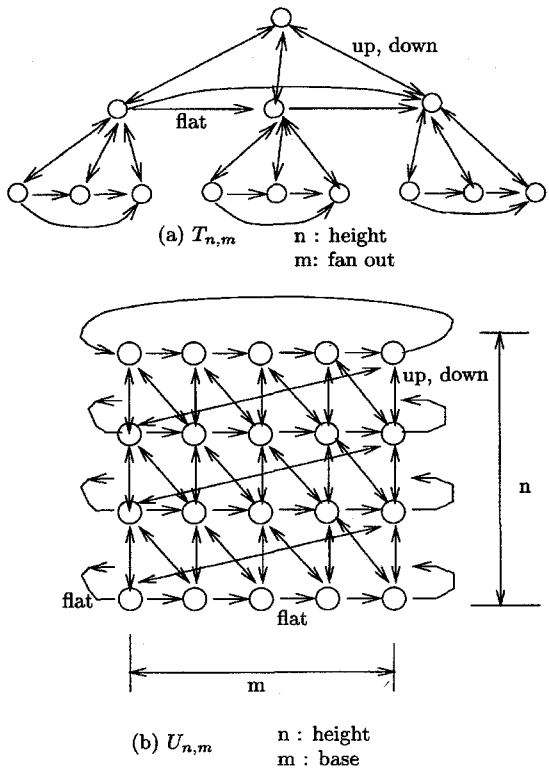



Fig. 12. Data sets $T_{n,m}$ and $U_{n,m}$.

techniques in the middle of the last decade. More detailed descriptions of those data sets can be found in the corresponding references.

## REFERENCES

1. L. Balbin and K. Ramamohanarao, A generalization of the differential approach to recursive query evaluation, *J. Logic Programming* 4(3):259–262 (1987).
2. F. Bancilhon, Naive evaluation of recursively defined relations, in: J. Mylopoulos and M. L. Brodie (Eds.), *On Knowledge Base Management Systems—Integrating Artificial Intelligence and Database Technologies*. Springer-Verlag, New York, 1985, pp. 165–178.
3. F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, Magic sets and other strange ways to implement logic programs, in: *Proceedings of the Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Boston, MA, March 1986, pp. 1–15.

4. F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query processing strategies, in: *Proceedings of the* 1986 *ACM SIGMOD International Conference of Management of Data*, Washington, DC, May 1986, pp. 16–52.
5. C. Beeri and R. Ramakrishnan, On the power of magic, *J. Logic Programming* 10(3):255–300 (1991).
6. J. Cai and R. Paige, Program derivation by fixed point computation, Research Report RC13947, IBM Research Division, T. J. Watson Research Center, Yorktown Height, NY, 1988.
7. S. Ceri, G. Gottlob, and L. Lavazza, Translation and optimization of logic queries: The algebraic approach, in: *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Aug. 1986, pp. 395–402.
8. S. Ceri, G. Gottlob, and L. Tanca, What you always wanted to know about datalog (and never dared to ask), *IEEE Trans. Knowledge Data Engrg.* 1(1):146–166 (1989).
9. S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases*, Springer-Verlag, New York, 1990.
10. S. Ceri and L. Tanca, Optimization of systems of algebraic equations for evaluating datalog queries, in: *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, Sept. 1987, pp. 31–41.
11. C.-L. Chang and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.
12. K. L. Clark, Negation as failure, in: H. Gallaire and J. Minker (Eds.), *Logic and Data Bases*, Plenum Press, New York, 1978, pp. 293–322.
13. S. Ganguly, R. Krishnamurthy, and A. Silberschatz, An analysis technique for transitive closure algorithms: A statistical approach, in: *Proceedings of the Seventh International Conference on Data Engineering*, 1991, pp. 728–735.
14. M. Kifer and E. L. Lozinskii, On compile-time query optimization on deductive databases by means of static filtering, *ACM Trans. Database Syst.* 15(3):385–426 (1990).
15. G. Kildall, A unified approach to global program optimization, in: *Proceedings of the ACM Symposium on Principles of Programming Languages*, Oct. 1973.
16. J. Kuittinen, O. Nurmi, S. Sippu, and E. Soisalon-Soininen, Efficient implementation of loops in bottom-up evaluation of logic queries, in: *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, Aug. 1990, pp. 372–379.
17. H. Lu, New strategies for computing the transitive closure of a database relation, in: *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, 1987, pp. 267–274.
18. J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman, Argument reduction by factoring, in: *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, 1989, pp. 173–182.
19. R. Ramakrishnan, D. Srivastava, and S. Sudarshan, Controlling the search in bottom-up evaluation, in: *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992, pp. 273–287.
20. R. Ramakrishnan, D. Srivastava, and S. Sudarshan, Rule ordering in bottom-up fixpoint evaluation of logic programs, *IEEE Trans. Knowledge Data Engrg.* 6(4):501–517 (1994).
21. H. Schmidt, W. Kiessling, U. Guntzer, and R. Bayer, Compiling exploratory and goal-directed deduction into sloppy delta iteration, *IEEE International Symposium on Logic Programming*, 1987, pp. 234–243.

22. D. Srivastava, S. Sudarshan, R. Ramakrishnan, and J. F. Naughton, Space optimization in deductive databases, *ACM Trans. Database Syst.* 20(4):472–516 (1995).
23. A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific J. Math.* 5:285–309 (1955).
24. J. D. Ullman, Implementation of logical query languages for databases, *ACM Trans. Database Syst.* 10(3):289–321 (1985).
25. J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 1, Computer Science Press, Rockville, MD, 1988.
26. J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vol. 2, Computer Science Press, Rockville, MD, 1989.
27. K. Y. Whang and S. B. Navathe, An extended disjunctive normal form approach for optimizing recursive logic queries in loosely coupled environments, in: *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, Brighton, Sept. 1987, pp. 275–287.

*Received 1 March 1996; revised 22 May 1996*