Efficient Implementation of Loops in Bottom-Up Evaluation of Logic Queries*

Juhani Kuittinen[1], Otto Nurmi[1], Seppo Sippu[2], and Eljas Soisalon-Soininen[1]

[1] *Department of Computer Science, University of Helsinki*
Teollisuuskatu 23, SF-00510 Helsinki, Finland

[2] *Department of Computer Science, University of Jyväskylä*
Seminaarinkatu 15, SF-40100 Jyväskylä, Finland

Abstract. We consider the efficient implementation of the bottom-up evaluation method for recursive queries in logic databases. In the bottom-up evaluation algorithms the non-mutually-recursive rules are evaluated in certain order, whereas the evaluation order within a set of the mutually recursive rules is free. However, significant savings in join operations can be achieved by arranging the mutually recursive rules appropriately. We present an algorithm for splitting the evaluation loop for mutually recursive rules into subloops and for determining the order in which the rules should be evaluated within a loop. The semi-naive evaluation algorithm is modified accordingly to gain advantage from the evaluation order and to work with the incremental relations ("deltas") appearing at different levels in the loop structure. The computation within a subloop is optimized by identifying loop-invariant factors in the rules to be evaluated. Using an experimental logic database system we demonstrate the usefulness of our algorithm in implementing datalog queries optimized by the "magic sets" and related term rewriting strategies.

*The work was supported by the Academy of Finland.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1. Introduction

Recursive data base queries expressed in datalog (function-free Horn clause programs) are most conveniently evaluated using the bottom-up (or forward chaining) evaluation method (see, e.g., [1, 2, 5, 14]). As the basic "naive" or "semi-naive" bottom-up method becomes inefficient when the query contains bound arguments, the usual approach is to preprocess the original query by means of some term rewriting strategy so that the bindings in the query literal can be used to restrict the set of database facts consulted [3, 4, 6]. Recently, it has been shown that it is possible to rewrite any safe datalog program as an equivalent datalog program so that the semi-naive bottom-up evaluation of the rewritten program is as efficient as the top-down evaluation of the original program [16].

The structure of the naive and semi-naive evaluation algorithm is determined by the "rule-goal graph", which represents the dependencies between the predicates and rules in the datalog program to be evaluated [13, 4]. The strongly connected components of this graph represent the maximal sets of mutually recursive predicates and rules. For each such component, the naive evaluation algorithm contains a loop that computes the values for the predicates in that component. The execution order between the loops is determined by the partial order between the strongly connected components, so that first comes the loop that implements the component whose predicates do not depend on predicates in the other components [13, 10, 7, 8, 4].

The values of the predicates within a component are computed in an incremental fashion: at each iteration every rule whose head and body contains a predicate belonging to the component is used to add data to the current value of the predicate in the head of the rule. The relations for the predicates are ini-

tially empty, and the computation will terminate when no new tuples can be added to any relation. In the naive evaluation the entire current values of the relations are used in computing the new values, whereas in the semi-naive evaluation differential techniques are used to avoid repeated computations [1, 2, 5, 4, 14].

The number of join operations performed in the evaluation depends directly on the number of iterations required by each loop. Thus, special attention should be paid to optimizing the loop structure of the evaluating algorithm.

Let us consider the following simple example program (where we have left out the arguments of the predicates).

1. $P_1 :- C.$
2. $P_1 :- P_2.$
3. $P_2 :- P_3.$
4. $P_3 :- P_4.$
5. $P_4 :- P_5.$
6. $P_5 :- P_1.$

The predicates $P_1, \ldots, P_5$ are mutually recursive. If the recursive rules $2 \ldots 6$ are evaluated in the order $\{2,3,4,5,6\}$ in the loop of the naive evaluation algorithm, the relation corresponding to P_2 remains empty until the fourth iteration. A much better way is to evaluate the rules in the reverse order $\{6,5,4,3,2\}$ in which P_2 gets its first tuples already in the first iteration.

The structure of the example program is quite simple — the rule-goal graph corresponding to the program has only one simple cycle. The structure of programs obtained by optimizing datalog queries by the "magic sets" and related methods tend to become complicated, and it is no longer a trivial task to choose a good evaluation order for the mutually recursive rules. In Section 2 we shall explain a method to determine an appropriate evaluation order for a set of mutually recursive rules.

In order to benefit of the evaluation order, the tuples obtained for a predicate during an iteration must immediately be used to compute new tuples for the other predicates during the same iteration. In the conventional semi-naive evaluation, the tuples are not available before the following iteration. The optimization aspect of using the just computed tuples immediately is shortly discussed in [7] and in [15, Exercise 12.7]. In Section 3 we shall show in detail how to manage with the incremental relations ("deltas") during the semi-naive evaluation so that we obtain maximal benefit from the evaluation order of the rules. This includes the identification of loop-invariant factors in the rules to be evaluated within a subloop; these factors are then moved outside the loop and computed at an outer level in the loop structure.

The strategy is used in an experimental logic database system being implemented at the University of Helsinki. Our experience shows that the number of join operations can be reduced even 50% by using the algorithm in connection with programs found in literature and the queries optimized by "magic sets" and related rewriting strategies (Section 4).

We assume the reader is familiar with the basic notions pertaining to datalog programs [4, 14].

2. Determining the Evaluation Order

Consider the datalog program

1. $P :- A.$
2. $P :- A, P, B.$
3. $Q :- S.$
4. $R :- Q, P.$
5. $R :- Q, R.$
6. $S :- C.$
7. $S :- C, Q, R.$

Here A, B, and C are base predicates. The rule-goal graph [4, 13] for this program is shown in Figure 1.

As suggested in [13, 4, 10], the rule-goal graph can be used to organize the naive and semi-naive evaluation of the program so that each maximal subset of mutually recursive predicates is evaluated as a whole. These maximal subsets are obtained by determining the strongly connected components [12] of the rule-goal graph. The components that are singleton sets represent base predicates and non-recursive rules; all the other components represent maximal sets of mutually recursive predicates and rules that are recursive with these predicates.

The components are sorted topologically [9] according to the partial order induced by the arcs of the rule-goal graph between the components. First come those components that have no incoming arcs. For example, one of the possible linear orders for the components of the graph in Figure 1 is:

$$\{A\}, \{1\}, \{B\}, \{P, 2\}, \{C\}, \{6\}, \{Q, R, S, 3, 4, 5, 7\}$$

(the rules are indicated by their numbers).

Each component representing a set of mutually recursive predicates gives rise to a loop of its own in the naive or semi-naive evaluation algorithm. In our example there are two loops: one for evaluating the

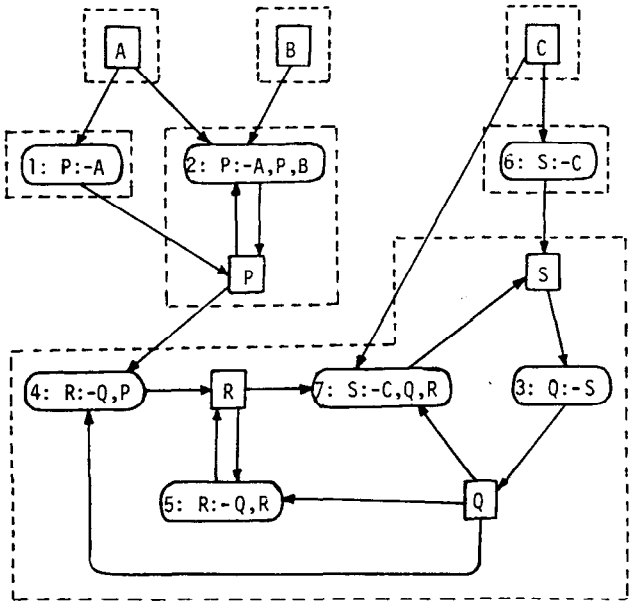

Fig. 1. A rule-goal graph. Square nodes denote predicate nodes and oval nodes denote rule nodes. The strongly connected components are enclosed by a broken line.

recursive rule 2 and the other for evaluating the set of mutually recursive rules $\{3,4,5,7\}$.

The order guarantees that a non-recursive rule is not evaluated until the rules for its subgoals have been evaluated. Moreover, the rules for those subgoals of a recursive rule r that are not mutually recursive with the head of r will be evaluated before evaluating any rule mutually recursive with r.

The strategy above does not impose any order between the rules within a single component. In the following we shall refine the strategy so that every component will eventually be totally ordered in a way that decreases the number of iterations needed in the loops and, accordingly, the number of relational operations needed when the rules in the loops are evaluated. In Section 3 we shall then show how to utilize this order in implementing the semi-naive evaluation algorithm.

Our strategy is to use *the same strategy recursively* for the components. The nodes corresponding to a non-singleton component do, of course, not have a topological order. Therefore we temporarily remove some edges from the component. These edges are chosen so that the component will in any case be split into smaller components that have a topological order, and so that the head predicate of the first rule in the obtained linear order will have rules outside the component, too. The strategy is then repeated recursively for the non-singleton strongly connected subcomponents. They will appear as nested loops in the final evaluation program. The details of the strategy is explained in the following.

Let C be a non-singleton component. Obviously, C must contain some predicate node to which an arc is coming from outside, i.e., from a node in some component preceding C in the linear order of the components. We call such a node an *entry node*. The only case in which a component may not have an entry node is that the program contains useless predicates that cannot derive a string of base literals (such predicates necessarily compute the empty set of tuples and should be eliminated from the program). In our example we find that P is an entry node of the component $\{P,2\}$ and that S is the only entry node of the component $\{Q,R,S,3,4,5,7\}$.

We split each non-singleton component into smaller components by removing some arcs between the nodes in the component. In order to make sure that the component will really be split, we simply remove all the arcs that are coming to the entry node from the other nodes of the component. (If there are several entry nodes, one of them is selected.) Thus, from the component $\{P,2\}$ we remove the arc $(2,P)$ and from the component $\{Q,R,S,3,4,5,7\}$ we remove the arc $(7,S)$. To each such component we then apply the algorithm for finding the strongly connected components, and sort these topologically. Notice that the component will always be split: at least the entry node will form a component of its own. In our example, the component $\{P,2\}$ is split into the components $\{P\}$, $\{2\}$, given in the linear order resulting from the topological sort. The component $\{Q,R,S,3,4,5,7\}$ is similarly split into the list of components $\{S\},\{3\},\{Q\},\{4\},\{R,5\},\{7\}$.

We repeat the above process of splitting components until all the remaining components are singletons. In the example, the process will terminate after the component $\{R,5\}$ is split into $\{R\}$, $\{5\}$. The result of the algorithm is a nested structure of linear orders:

$$A,1,B,(P,2),C,6,(S,3,Q,4,(R,5),7).$$

Each substructure $(\ldots)$ will give rise to a loop of its own in the semi-naive evaluation algorithm. Inside the loop the rules listed in the substructure will be evaluated in the order indicated. Observe that each rule will receive tuples from at least some of the preceding predicate nodes (or they will remain empty in the result) and that each substructure begins with the entry node of the corresponding component. As the entry node, by definition, receives tuples from some preceding component, these tuples can thus be made available to the next loop at the earliest possible convenience.

In generating the semi-naive evaluation algorithm from the order structure, the predicate nodes will no

longer be needed. Thus we may represent the above order structure more succinctly as

$$1,(2),6,(3,4,(5),7)$$

which is the same as

$$P :- A,\ (P :- A, P, B),\ S :- C,$$
$$(Q :- S,\ R :- Q, P,\ (R :- Q, R),\ S :- C, Q, R).$$

3. Implementing the Order Structure

The conventional semi-naive evaluation (see, e.g, [14]) does not use the tuples computed for a predicate at the current iteration when the predicate is used to compute tuples for other predicates in the same iteration. Therefore, the order of the rules in a loop has no effect in the number of iterations needed. In this section we shall show how to manage with the incremental relations, i.e. Δ's, in the loops of a program implementing the semi-naive evaluation so that the termination of the loop can be correctly determined and, apart from that, the tuples found during an iteration can immediately be used in other rules to generate new tuples. The implementation also includes the use of incremental relations in the case of nested loops. (In the conventional implementation, there are loops only at one level.)

We shall now explain the details of the implementation of the loop at level $l \geq 1$ of the loop structure determined by the algorithm of Section 2. Let $\mathcal{S}$ be the set of predicates that appear in the rules to be evaluated in the loop and in the higher level loops enclosed by it. We divide $\mathcal{S}$ into two sets, $\mathcal{P}$ and $\mathcal{Q}$, where $\mathcal{P}$ is the set of predicates that appear in the head of some rule to be evaluated in the loop and $\mathcal{Q}$ is the set of other predicates.

For each head predicate $P \in \mathcal{P}$ we introduce two temporary incremental relations, denoted by $\Delta_l P$ and $\Delta'_l P$. The invariant for $\Delta_l P$ is that it is the set of tuples inserted into P during the most recent completed iteration of the loop. The invariant for $\Delta'_l P$ is that it is the set consisting of $\Delta_l P$ and of the tuples inserted into P earlier at the present iteration. Notice that the invariant for $\Delta_l P$ is the same as that for the incremental relation in the conventional implementation. The relation $\Delta_l P$ will be used to control the termination of the loop, and $\Delta'_l P$ will be used for the termination control and to compute new tuples in the loop. We always have $\Delta_l P \subseteq \Delta'_l P \subseteq P$.

Before the loop, the relation $\Delta_l P$ is initialized to $\emptyset$ and $\Delta'_l P$ to $\Delta'_{l-1} P$ or, if $l = 1$, to P. In the beginning of each iteration, $\Delta_l P$ and $\Delta'_l P$ are initialized to contain the same set of tuples, i.e., the tuples inserted into P during the most recent iteration. These tuples are obtained as the difference $\Delta'_l P - \Delta_l P$.

During the iteration, new tuples are inserted into $\Delta'_l P$ but not into $\Delta_l P$. Thus the termination condition for the loop, "no new tuples are found during the iteration", is $\Delta'_l P = \Delta_l P$ for all $P \in \mathcal{P}$.

In the conventional implementation new tuples are computed by using the whole relations and the increments from the most recent iteration. More specifically, let

$$P :- S_1, S_2, \ldots, S_m \tag{1}$$

be a rule to be evaluated in the loop, and $k_1, k_2, \ldots, k_n$ be the indices of the predicates of $\mathcal{P}$ in its body. In the conventional implementation, the code generated for rule (1) consists of the statements

insert EVALUATE$(P :- S_1, \ldots, S_{k_i-1}, \Delta_1 S_{k_i},$
$S_{k_i+1}, \ldots, S_m) - P$
into P **and** $\Delta_1 P$

for $i = 1, \ldots, n$, where EVALUATE(r) denotes the tuples obtained by evaluating the argument rule r (without the assignment to the head).

In our modified implementation we shall use the order of rules as computed by the algorithm of Section 2. We must use the incremental relations already in the iteration in which increments are found. As defined above, the Δ'_l relations contain these increments and those from the most recent completed iteration. On the first loop level we therefore simply replace the relations $\Delta_1 S_{k_i}$ of the conventional implementation by the relations $\Delta'_1 S_{k_i}$.

Let us assume that we are generating code for the rule

$$P :- S_1, S_2, \ldots, S_m \tag{2}$$

in a loop of level $l > 1$. Let $k_1, k_2, \ldots, k_n$ be the indices of the predicates in the body of (2) for which S_{k_i} is the head of a rule in the present loop, in the loops enclosed by it (level$> l$), or in the loops that enclose it (level$< l$). Let $\max_{k_i}$ denote l if S_{k_i} is the head of a rule to be evaluated in the present loop or a loop enclosed by it and, otherwise, the level of the highest loop that enclose the present one and where S_{k_i} appears as a head predicate. The rule (2) could now be implemented by the statements

insert EVALUATE$(P :- S_1, S_2, \ldots, S_{k_i-1}, \Delta'_{\max_{k_i}} S_{k_i},$
$S_{k_i+1}, \ldots, S_m) - P$
into $P, \Delta'_1 P, \Delta'_2 P, \ldots,$ and $\Delta'_l P,$

for $i = 1, \ldots, n$.

```
    P := ∅; Q := ∅; R := ∅; S := ∅;
P :- A :
    insert EVALUATE(P :- A) - P into P;
P :- A, P, B :
    Δ'1P := P; Δ1P := ∅;                    {Initialize Δ's.}
    repeat
        Δ1P := Δ'1P - Δ1P;                  {Tuples from the most recent iteration.}
        Δ'1P := Δ1P;                        {Tuples from this and the most recent iteration.}
        insert EVALUATE(P :- A, Δ'1P, B) - P into P and Δ'1P;
    until Δ'1P = Δ1P;                       {No more new tuples.}
S :- C :
    insert EVALUATE(S :- C) - S into S;

    for X ∈ {Q, R, S} do begin Δ'1X := X; Δ1X := ∅ end;
                                            {Initialize Δ's for all head predicates
                                             of the rules in the following loop}
    repeat
        for X ∈ {Q, R, S} do Δ1X := Δ'1X - Δ1X;
                                            {Tuples from the most recent iteration.}
        for X ∈ {Q, R, S} do Δ'1X := Δ1X; {Tuples from this and the most recent iteration.}
    Q :- S :
        insert EVALUATE(Q :- Δ'1S) - Q into Q and Δ'1Q;
    R :- Q, P :
        insert EVALUATE(R :- Δ'1Q, P) - R into R and Δ'1R;
    R :- Q, R :
        insert EVALUATE(R :- Δ'1Q, R) - R into R and Δ'1R;
                                            {The statements moved from the following loop.}
        Δ'2R := Δ'1R; Δ2R := ∅;              {Initialize Δ's for the inner loop.}
        repeat
            Δ2R := Δ'2R - Δ2R;
            Δ'2R := Δ2R;
            insert EVALUATE(R :- Q, Δ'2R) - R into R, Δ'1R, and Δ'2R
                                            {Update Δ'1R (the outer loop), too.}
        until Δ'2R = Δ2R;
    S :- C, Q, R :
        insert EVALUATE(S :- C, Δ'1Q, R) - S into S and Δ'1S;
        insert EVALUATE(S :- C, Q, Δ'1R) - S into S and Δ'1S;

    until Δ'1Q = Δ1Q and Δ'1R = Δ1R and Δ'1S = Δ1S.
```

Fig. 2. The modified semi-naive evaluation program for the example of Section 2. The order structure of the program is $P :- A,\ (P :- A, P, B),\ S :- C,\ (Q :- S,\ R :- Q, P,\ (R :- Q, R),\ S :- C, Q, R)$.

The incremental relations $\Delta'_{\max_{k_i}} S_{k_i}$ remain constant inside the loop whenever $\max_{k_i} < l$. Therefore, these statements can more efficiently be computed outside the loop. They are moved onto the level where $\Delta'_{\max_{k_i}} S_{k_i}$ changes, that is, onto the level $\max_{k_i}$. On that level, the statements are placed immediately before the loop where (2) is to be evaluated.

In conclusion, a rule is, in general, evaluated by several **insert** statements of the above type. The level on which a statement is placed is the same as the index of the Δ' on the right hand side of the argument of the corresponding EVALUATE function. The relations into which the new tuples are inserted are the head of the rule and all lower level Δ' relations for the head. The following program fragment demonstrates the initialization of the incremental relations and the overall structure of a loop.

```
The statements moved from the loop to level l - 1;
for X ∈ P do begin Δ'lX := Δ'l-1X; ΔlX := ∅ end;
repeat
    for X ∈ P do ΔlX := Δ'lX - ΔlX;
    for X ∈ P do Δ'lX := ΔlX;
    The code for the rules and the subloops in the
    order indicated by the algorithm of Section 2;
until Δ'lX = ΔlX for all X ∈ P.
```

The code for the example program of Section 2 is shown in Figure 2.

4. Experimental Results

In this section we present experimental results obtained by applying our evaluation strategy to the "nonlinear same generation problem" [6]. This problem is expressed as how to compute the predicate *Query*, when the datalog program is:

$$sg(X,Y) :- flat(X,Y).$$
$$sg(X,Y) :- up(X,X_1), sg(X_1,X_2), flat(X_2,Y_2), sg(Y_2,Y_1), down(Y_1,Y).$$
$$Query(Y) :- sg(a,Y).$$

Here a is a constant and *flat*, *up*, and *down* are base predicates. Figure 3 shows two samples of data for the predicates *flat*, *up*, and *down*. We shall present results for data A_n with $n = 4, 6, 8, 10$ and for data B_n with $n = 4, 16, 64, 256$.

First we consider our evaluation strategy in connection with the magic sets optimization [3, 6]. Using "supplementary magic sets" [6], the above program is rewritten as:

1. $magic\text{-}sg(a)$.
2. $supmagic_2(X,X_1) :- magic\text{-}sg(X), up(X,X_1)$.
3. $supmagic_3(X,X_2) :- supmagic_2(X,X_1), sg(X_1,X_2)$.
4. $supmagic_4(X,Y_2) :- supmagic_3(X,X_2), flat(X_2,Y_2)$.
5. $sg(X,Y) :- magic\text{-}sg(X), flat(X,Y)$.
6. $sg(X,Y) :- supmagic_4(X,Y_2), sg(Y_2,Y_1), down(Y_1,Y)$.
7. $magic\text{-}sg(X_1) :- supmagic_2(X,X_1)$.
8. $magic\text{-}sg(Y_2) :- supmagic_4(X,Y_2)$.
9. $Query(Y) :- sg(a,Y)$.

For simplicity, we have left out the adornment (bf) of the predicates *sg* and *magic-sg*.

In the conventional semi-naive evaluation, rules 1 to 9 are evaluated in the order

$$1,(2,3,4,5,6,7,8),9,$$

where the order within the strongly connected component $\{2,3,\ldots,8\}$ is irrelevant (new tuples obtained for a predicate during an iteration are not used to compute new tuples for other relations until succeeding iterations).

The results for conventional semi-naive evaluation are shown in Table 1. The column "# tuples" gives the total number of tuples in the base relations *up*, *flat*, and *down*. The column "# iterations" gives the number of iterations of the loop for the strongly connected component $\{2,3,\ldots,8\}$. The column "# joins" gives the total number of joins performed during the evaluation. The column "join size" gives the total number of tuples in the input and output relations of all the join operations.

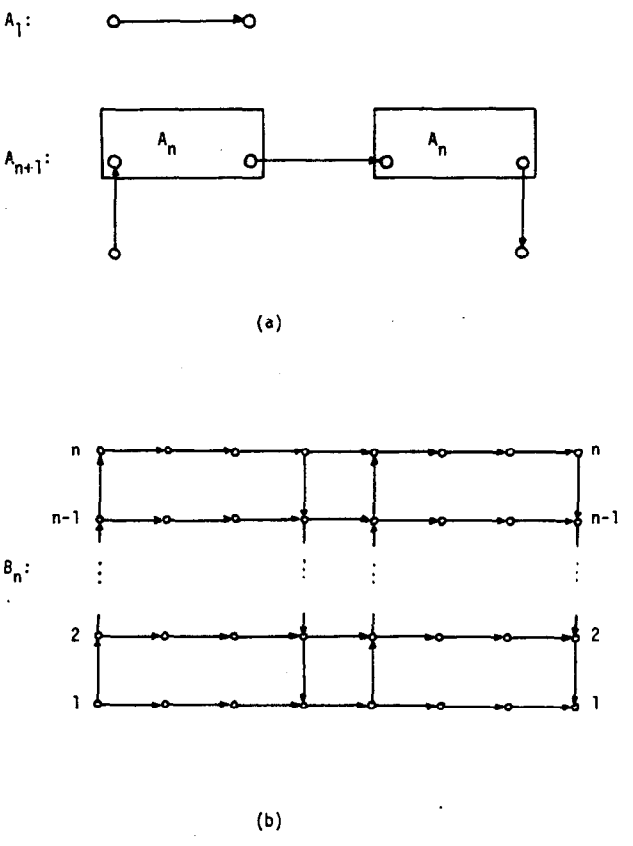

Fig. 3. Two sets of test data for the nonlinear same generation problem. The arrows going up denote tuples in the base relation *up*, the arrows going to the right denote tuples in the base relation *flat*, and the arrows going down denote tuples in the base relation *down*. Data A_n is defined inductively on n.

Table 2 shows the results for semi-naive evaluation implemented using our strategy. The results of Table 2 have been obtained by using the evaluation order

$$1,(2,7,5,(3,4,6),8),9.$$

This is the order produced by the algorithm given in Section 2. In Table 2, the subcolumn "outer" gives the number of iterations of the loop for $(2,7,5,(\ldots),8)$ and the subcolumn "inner" gives the number of iterations of the loop for $(3,4,6)$.

Comparing the figures in Table 1 with those in Table 2 indicates that the appropriate ordering of the evaluation of a set of mutually recursive rules may lead to a significant reduction in the number of joins as well as in the number of tuples involved in joins.

Next we consider our evaluation strategy in connection with the "regular envelopes" optimization presented in [11]. As regards the amount of database facts consulted, this strategy is less efficient than the magic sets strategy because the restricting "envelopes" computed may be larger than the true magic sets. However, the predicates computing these envelopes are never mutually recursive with the other predicates, so that their computation becomes effectively a preprocessing task.

Using regular envelopes, our example program is rewritten as:

1. $in\text{-}sg(a)$;
2. $in\text{-}sg(X_1) :- in\text{-}sg(X), up(X, X_1)$.
3. $in\text{-}sg(Y_2) :- in\text{-}sg(X), flat(X, Y_2)$.
4. $in\text{-}sg(Y_2) :- out\text{-}sg(X_2), flat(X_2, Y_2)$.
5. $out\text{-}sg(Y) :- in\text{-}sg(X), flat(X, Y)$.
6. $out\text{-}sg(Y) :- out\text{-}sg(X_2), flat(X_2, Y)$.
7. $out\text{-}sg(Y) :- out\text{-}sg(Y_1), down(Y_1, Y)$.
8. $sg(X, Y) :- in\text{-}sg(X), flat(X, Y)$.
9. $sg(X, Y) :- in\text{-}sg(X), up(X, X_1), sg(X_1, X_2), flat(X_2, Y_2), sg(Y_2, Y_1), down(Y_1, Y)$.
10. $Query(Y) :- sg(a, Y)$.

The algorithm of Section 2 produces the evaluation order

$$1, (2, 3, 5, (7, 6), 4), 8, (9), 10.$$

Thus the computation of *sg* will only begin after the "envelope predicates" *in-sg* and *out-sg* have been precomputed in the loop for $(2, 3, 5, (\ldots), 4)$. Table 3 shows the results for this order. The subcolumn "1st outer" gives the number of iterations of the loop for $(2, 3, 5, (\ldots), 4)$, the subcolumn "inner" the number of iterations of the loop for $(7, 6)$ and the subcolumn "2nd" the number of iterations of the loop for (9).

Comparing the figures in Table 3 with those in Table 2 shows how the loop structure resulting from the rewriting of the program in the optimization phase may affect the efficiency of the evaluation. The fact that the "envelope predicates" *in-sg* and *out-sg* are evaluated in a loop separate from that for *sg* has resulted in significant savings in the number of joins. On the other hand, the total number of tuples involved in joins is somewhat larger; this reflects the fact that the regular envelope computed by *in-sg* and *out-sg* is not so tight as the magic set computed by *magic-sg*.

data	# tuples	# iterations	# joins	join size
A_4	58	51	459	3,747
A_6	250	219	1,971	68,287
A_8	1,018	891	8,019	1,131,599
A_{10}	4,090	3,579	32,211	18,282,255
B_4	80	26	234	2,735
B_{16}	344	50	450	23,255
B_{64}	1,400	146	1,314	275,255
B_{256}	5,624	530	4,770	4,001,975

Table 1. Conventional semi-naive implementation of the magic sets optimized query.

data	# tuples	# iterations		# joins	join size
		outer	inner		
A_4	58	11	22	190	1,632
A_6	250	47	94	814	27,062
A_8	1,018	191	382	3,310	439,054
A_{10}	4,090	767	1,534	13,294	7,060,590
B_4	80	6	14	120	1,365
B_{16}	344	18	43	369	16,380
B_{64}	1,400	66	163	1,401	239,976
B_{256}	5,624	258	643	5,529	3,743,640

Table 2. Modified semi-naive implementation of the magic sets optimized query.

data	# tuples	# iterations			# joins	join size
		1st outer	inner	2nd		
A_4	58	12	30	4	149	2,487
A_6	250	48	126	6	505	29,163
A_8	1,018	192	510	8	1,869	400,323
A_{10}	4,090	768	2,046	10	7,265	6,052,971
B_4	80	5	23	3	97	2,720
B_{16}	344	17	107	3	313	27,920
B_{64}	1,400	65	443	3	1,177	376,400
B_{256}	5,624	257	1,787	3	4,633	5,733,200

Table 3. Modified semi-naive implementation of the regular envelopes optimized query.

5. Conclusion

We have generalized the semi-naive bottom-up evaluation strategy so as to gain advantage of the entire loop structure of the datalog program being evaluated. First, we presented an algorithm for determining for any datalog program a nested structure of linear orders in which each substructure represents a sequence of rules that will be evaluated within a loop of their own in the evaluation algorithm. The algorithm operates with the rule-goal graph by repeatedly splitting strongly connected components into subcomponents and determining an appropriate linear order between these subcomponents.

Secondly, we showed how any order structure can be implemented as a modified semi-naive algorithm that can manage with nested loops and incremental relations corresponding to different levels in the order structure. This algorithm includes the optimization that loop-invariant factors in rule bodies are identified and pushed onto that level in the loop structure in which their values really change.

Finally, we presented experimental results obtained by using the above method to evaluate a non-linearly recursive datalog program optimized by "magic sets", and one optimized by a similar rewriting strategy but leading to a different loop structure. Significant savings in join operations were achieved when compared to the conventional implementation of the semi-naive algorithm.

References

1. I. Balbin, and K. Ramamohanarao, A differential approach to query optimization in recursive deductive databases. TR–86/7, Dept. of Computer Science, University of Melbourne, 1986.

2. F. Bancilhon, Naive evaluation of recursively defined relations. In: *On Knowledge Base Management Systems—Integrating Artificial Intelligence and Database Technologies,* (Brodie and Mylopoulos, eds), Springer-Verlag, 1986, 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: *Proc. 5th ACM Symp. on Principles of Database Systems,* 1986, pp 1–15.

4. F. Bancilhon, and R. Ramakrishnan, An amateur's introduction to recursive query processing strategies. In: *Proc. ACM SIGMOD'86, Internat. Conf. on Management of Data,* 1986, pp 16–52.

5. R. Bayer, Query evaluation and recursion in deductive database systems. Unpublished memorandum, Technical University of Munich, 1985.

6. C. Beeri, and R. Ramakrishnan, On the power of magic. In: *Proc. 6th ACM Symp. on Principles of Database Systems,* 1987, pp 269–283.

7. S. Ceri, G. Gottlob, and L. Lavazza, Translation and optimization of logic queries: the algebraic approach. In: *Proc. 12th Internat. Conf. on Very Large Data Bases,* 1986, pp 395–402.

8. S. Ceri, and L. Tanca, Optimization of systems of algebraic equations for evaluating datalog queries. In: *Proc. 13th Internat. Conf. on Very Large Data Bases,* 1987, pp 31–41.

9. D. E. Knuth, *The Art of Computer Programming. Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1968.

10. K. Morris, J. D. Ullman, and A. Van Gelder, Design overview of the NAIL! system. In: *Proc. 3rd Internat. Conf. on Logic Programming,* London, 1986, Springer-Verlag, pp 554–568.

11. S. Sippu, and E. Soisalon-Soininen, An optimization strategy for recursive queries in logic databases. In: *Proc. 4th Internat. IEEE Conf. on Data Engineering,* 1988, pp 470–477.

12. R. Tarjan, Depth first-search and linear graph algorithms. *SIAM J. Comput.* **1** (1972), 146–160.

13. J. D. Ullman, Implementation of logical query languages for databases. *ACM Trans. Database Syst.* **10** (1985), 289–321.

14. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, vol. I,* Computer Science Press, 1988.

15. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, vol. II: The New Technologies,* Computer Science Press, 1989.

16. J. D. Ullman, Bottom-up beats top-down for datalog. In: *Proc. 8th ACM Symp. on Principles of Database Systems,* 1989, pp 140–149.