Bottom-Up Method for Processing Recursive Sets of Rules

Velko Iltchev

Abstract: *The paper proposes a semi-naive method for processing recursive loops in the dependency graph for a given query. The process goes through two phases. During the expand phase answers are generated using translation to base conjunctions. Entries in recursive predicates with undistinguished arguments are also stored in the database for further processing. During the shrink phase the occurrences of recursive predicates in rule-bodies are replaced with the answers already generated during the expand phase. Thus, the whole rule-body becomes a base conjunction, which generates new answers. The proposed method is suitable for queries with bound arguments. It reduces unnecessary computations and generates only facts relevant to the given query.*

Key words: *deductive databases, logic programming, knowledge-based systems, algorithms*

INTRODUCTION

In the bottom-up methods for rules processing in deductive database systems the inference engine starts with the facts and applies the rules to generate new facts. This allows the usage of such powerful relational algebra operations like selection, projection, join, etc., which makes the information retrieval *set-at-a-time*-oriented.

As facts are generated, they are checked against the query predicate for a match. This leads to the main disadvantage of these methods which is that all possible facts are generated, most of them irrelevant to the particular query. This is very inefficient for large databases. Therefore, a search strategy to generate only the facts relevant to the query should be used.

One of the first methods, which applies such a strategy, was proposed in [4]. It is based on interpretation of the predicates as functions. But this method is not general, because it requires linearity of the rules.

The later methods use rewriting techniques. They exploit the goal structure, to transform an inefficient program into a "smart" one, written in the same language. Such methods are: the "Magic Sets Transformation Method" and the "Counting Method" presented in [1], as well as the "Static Filtering Method" described in [6].

This paper aims to propose a new method for recursive rules processing, which achieve the same optimization effect, as the methods mentioned above, but without rewriting. The proposed strategy uses iteration to process the recursion in the rules (or set of rules), but a recursive approach is also feasible. The deduction process goes through two phases: *Expand phase* and *Shrink phase*. At each iteration step of these two phases a differential (a quantity of new facts leading to the answer) is computed and stored in the predicate's temporal relation for future use. Only the differential computed during the previous step is used to compute the differential for the next step, which means - the proposed method is semi-naive. This reduces the amount of repeatable computations. Moreover, an additional selection implemented by computation of the differentials in the shrink phase reduces considerable the amount unnecessary facts, transmitted towards the next iteration steps.

LAYOUT

1. TERMINOLOGY.

According to the definitions given in [2]:

- *extensional database* (*EDB*) is a finite set of positive ground facts. It represents the data that are effectively stored in the relations of a relational database;

- *intensional database* (*IDB*) is a finite set of derived relations defined through a set of rules;

- *database predicate* (*dbp*) - it corresponds to a relation stored in the *EDB*, however,

its arguments correspond to the attributes of that relation;

- *rule defined predicate* (*rdp*) - it consists of a head, which is one positive literal and a body, formed by several *dbp*s and *rdp*s separated by commas. The head and the body are separated by ":-". The temporal results of processing this predicate are stored in separate temporal relation in the *EDB*;

- *base conjunction* (*bc*) - it is a sequence of *dbp*s. It can be part of a rule body.

Algorithms for translation of base conjunctions into relational algebra expressions have been described in many papers. One of them is [7]. Therefore these algorithms will not be covered here.

According to the definitions given in [2]:

- *sideways information passing* (*sip*) describes how bindings of arguments are passed from rule's head to rule's body or to other rules. Informally, for a rule of a program, a *sip*-graph represents the order in which the predicates in rule's body should be evaluated.

- *adornment* for an *n*-ary predicate *p* is a string *a* of length *n* on the alphabet *{b,f}*, where *b* stands for *bound* and *f* stands for *free*. The arguments order is fixed. An adorned occurrence of the predicate, p^a, corresponds to a computation of the predicate with some arguments bound to constants, and the other arguments free.

The method for generation of the dependency graph and the table structure for storing it proposed by the author of this paper in [5], is suitable for recursive sets of rules also. The algorithm for detecting loops in such a table structure is trivial. Therefore it will not be discussed here. Each loop in the dependency graph is caused by a recursion in a rule or set of rules. The method for processing such a recursion will be described in the next chapter.

2. THE RECURSION PROCESSING METHOD.

As mentioned in the introduction, the deduction process goes through two phases: *Expand phase* and *Shrink phase*. At each iteration step of these two phases a differential is computed and stored into the predicate's temporal relation for further use. The algorithms for computing these differentials are different for the different phases. In order to explain them the following definitions have to be additionally introduced:

A predicate argument is distinguished if it has a restricted range of values. This happens if either of the following conditions holds:

- the argument is a constant;
- the argument is bound by the adornment;
- the argument appears in a *dbp* that has at least one distinguished argument.

According to the last condition - a *dbp* is distinguished if it has at least one distinguished argument.

Let's have as an example the following rule set, which defines the classical "*same generation cousin*" relationship, represented through the predicate *sgc*:

r1: sgc(X,Y) :- eq(X,Y).
r2: sgc(X,Y) :- par(X1,X), sgc(X1,Y1), par(Y1,Y).

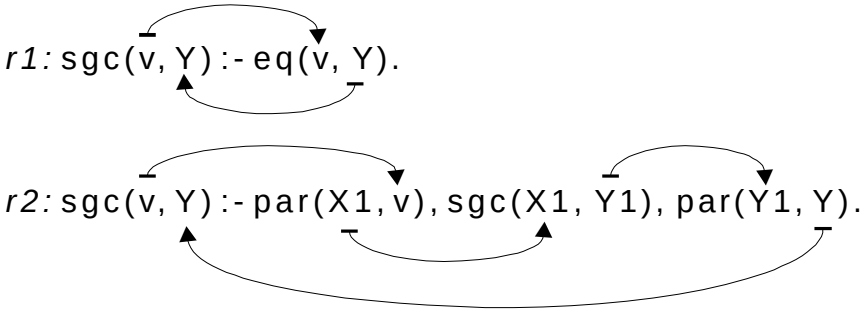
The facts in this example are described through the predicate par(X,Y), which means that *X* is parent of *Y*. The information about par(X,Y) is stored in the *EDB* as a relational table with attributes {X,Y}.

The predicate eq(X, Y), can be implemented through a biuld-in function or through a relation in the *EDB*, with the following tuples:

EQ = {<a,a>, <b,b>, <c,c>, <d,d>, …, <z,z>}

Let's have the query:
:- sgc(v,Y).

It causes the argument bindings, represented through the following *sip*-graphs:



Apparently the *X1* argument of the *sgc* predicate in the body of *r2* is distinguished, because it is bound to *X1* of the first *par* predicate. This argument becomes a restricted range of values generated by the selection on the *PAR* relation of the *EDB* where the attribute Y = 'v'. But the second argument *Y1* of the *sgc* predicate is undistinguished.

In such cases (recursive predicate with undistinguished arguments) I propose to generate a quantity of tuples for the recursive predicate, with distinguished arguments substituted with the values from the restricted range and undistinguished arguments substituted with the *null* value. Each of these tuples will be called from now on *request* and will also be stored into the recursive predicate's temporal table for further processing.

An *expand phase differential* is the result of *sip* of the requests, generated on the previous iteration step. This differential is union of two quantities:

- quantity of facts generated through propagating to base conjunctions. One possible way to generate these facts is to rewrite the recursive rules, substituting the recursive predicates with base conjunctions. Another way, proposed by the author of this paper in [5], is to iterate through the rule-dependency graph thus generating these facts from the base conjunctions toward the recursive predicate.
- quantity of new requests generated through *sip*, where the distinguished arguments come from a base conjunction build only from the distinguished *dbp*s in the rule body. The undistinguished arguments are substituted with the *null* value.

The *expand phase differential* will be abbreviated from now on as $_{exp}D^{StepNo}$, where *StepNo* is the number of the current iteration step.

The expand phase continues until the subsequent *expand phase differential* becomes an empty set.

The *shrink phase differential* is computed by replacing the occurrences of recursive predicates in rule-bodies with the answers already generated during the expand phase. Thus, the whole rule-body becomes a base conjunction, which generates new answers. This replacement will be called from now on *reverse substitution*. *Expand phase differentials* will be used subsequent for reverse substitutions during the shrink phase.

The *shrink phase differential* will be abbreviated from now on as $_{shr}D^{StepNo}$, where *StepNo* is the number of the current iteration step.

The shrink phase continues until the subsequent *shrink phase differential* becomes an empty set.

The SQL-Query for the subsequent *shrink phase differential* is composed as follows:

SELECT DISTINCT π <arguments list>
FROM $exp D^{MaxExpSteps-CurrentShrinkStep}$ ED, BaseConjunction BC
WHERE ED.<bound arguments> **IN**
(
SELECT π <bound arguments list>
FROM $exp D^{MaxExpSteps-CurrentShrinkStep-1}$
WHERE <free arguments> **IS NULL**
)
AND <base conjunction's join-condition>

UNION

SELECT <whole tuples>
FROM $exp D^{MaxExpSteps-CurrentShrinkStep}$
WHERE <free arguments> **IS NOT NULL**

To remove repeatable values the *DISTINCT* operator has to be used in the *SELECT* clause. The nested *SELECT* operation is used to separate only those facts, which correspond to the requests generated during the subsequent step of the expand phase. Recall that these requests have free arguments substituted with the *NULL*-value. Only the this way selected facts have to be used for *reverse substitution* in the base conjunction (look after the *AND* operator). This eliminates unnecessary computations not only outside the *decision cone*, but also inside it. Other methods do that only outside the *decision cone*.

EXAMPLE

Let's have as data for *PAR*:
PAR = {<a,f>, <b,g>, <c,g>, <c,h>, <d,i>, <e,i>, <e,j>, <f,k>, <f,l>, <g,l>, <h,m>, <h,n>, <i,n>, <i,o>, <j,o>, <k,p>, <k,q>, <l,q>, <l,r>, <m,r>, <m,s>, <n,s>, <o,t>, <o,u>, <q,v>, <q,v>, <r,x>, <s,x>, <s,y>, <t,y>, <u,z>}
which represents a recursive data structure,
and let's have the query:
:- sgc(v,Y).

Method's application:

// Expand phase

$exp SGC^0 = Ø$.

$exp D^1$ = {<v,v>, <q,q>, <v,w>, <q,null>}.
$exp SGC^1 = exp SGC^0 \cup exp D^1$ = {<v,v>, <q,q>, <v,w>, <q,null>}.

$exp D^2$ = {<k,k>, <l,l>, <q,p>, <q,r>, <k,null>, <l,null>}.
$exp SGC^2 = exp SGC^1 \cup exp D^2$ = {<v,v>, <q,q>, <v,w>, <q,null>, <k,k>, <l,l>, <q,p>, <q,r>, <k,null>, <l,null>}.

$exp D^3$ = {<f,f>, <g,g>, <k,l>, <l,k>, <f,null>, <g,null>}.
$exp SGC^3 = exp SGC^2 \cup exp D^3$ = {<v,v>, <q,q>, <v,w>, <q,null>, <k,k>, <l,l>, <q,p>, <q,r>, <k,null>, <l,null>, <f,f>, <g,g>, <k,l>, <l,k>, <f,null>, <g,null>}.

$_{exp}D^4$ = {<a,a>, <b,b>, <c,c>, <g,h>, <a,null>, <b,null>, <c,null>}.

$_{exp}SGC^4$ = SGC^3 U D^4 = {<v,v>, <q,q>, <v,w>, <q,null>, <k,k>, <l,l>, <q,p>, <q,r>, <k,null>, <l,null>, <f,f>, <g,g>, <k,l>, <l,k>, <f,null>, <g,null>, <a,a>, <b,b>, <c,c>, <g,h>, <a,null>, <b,null>, <c,null>}.

$_{exp}D^5$ = Ø. // End of the expand phase.

// Shrink phase

$_{shr}SGC^0$ = Ø.

$_{shr}D^1$ = {<g,h>}.
$_{shr}SGC^1$ = $_{shr}SGC^0$ U $_{shr}D^1$ = {<g,h>}.

Reverse substitution (only with the subsequent expand phase differential $_{exp}D^3$):
sgc(X,Y) :- par(X1,X), sgc(g,h), par(Y1,Y).

$_{shr}D^2$ = {<l,m>, <l,n>} U {<k,l>, <l,k>} = {<l,m>, <l,n>, <l,k>, <k,l>}.
$_{shr}SGC^2$ = $_{shr}SGC^1$ U $_{shr}D^2$ = {<g,h>, <l,m>, <l,n>, <k,l>, <l,k>}.

Reverse substitution (only with the subsequent expand phase differential $_{exp}D^2$):
sgc(X,Y) :- par(X1,X), sgc(l,m), par(Y1,Y).
sgc(X,Y) :- par(X1,X), sgc(l,n), par(Y1,Y).
sgc(X,Y) :- par(X1,X), sgc(k,l), par(Y1,Y).
sgc(X,Y) :- par(X1,X), sgc(l,k), par(Y1,Y).

After removing the repeatable values through *SELECT DISTINCT*, the $_{shr}D^3$ becomes:
$_{shr}D^3$ = {<q,r>, <q,s>, <r,r>, <r,s>, <p,q>, <p,r>, <q,p>, <q,q>, <r,p>, <r,q>} U {<k,k>, <l,l>, <q,p>, <q,r>}.
After selection because of <q,null>:
$_{shr}D^3$ = {<q,r>, <q,s>, <q,p>, <q,q>}.
Here it becomes visible that facts like <r,s>, <l,l>, <r,q>, etc. which lie inside the *decision cone*, are also removed. This reduces considerable the amount unnecessary facts, transmitted towards the next iteration steps.
$_{shr}SGC^3$ = $_{shr}SGC^2$ U $_{shr}D^3$ = {<g,h>, <l,m>, <l,n>, <k,l>, <l,k>, <q,r>, <q,s>, <q,p>, <q,q>}.

Reverse substitution (only with the subsequent expand phase differential $_{exp}D^1$):
sgc(X,Y) :- par(X1,X), sgc(q,r), par(Y1,Y).
sgc(X,Y) :- par(X1,X), sgc(q,s), par(Y1,Y).
sgc(X,Y) :- par(X1,X), sgc(q,p), par(Y1,Y).
sgc(X,Y) :- par(X1,X), sgc(q,q), par(Y1,Y).

$_{shr}D^4$ = {<v,x>, <w,x>, <v,y>, <w,y>, <v,w>} U {<v,v>, <q,q>, <v,w>}.

After a final selection with the first argument bound to 'v' the $_{shr}D^4$ becomes:
$_{shr}D^4$ = {<v,x>, <v,y>, <v,w>, <v,v>}.

This is the answer to the given query.

CONCLUSIONS AND FUTURE WORK

The proposed method achieves the same optimization effect like the methods based on rewriting techniques, because it builds the same *decision cone* during the deduction process. Moreover, the additional selection implemented by computation of the differentials in the *shrink phase* reduces considerable the amount unnecessary facts, transmitted towards the next iteration steps. This leads to elimination of unnecessary computations not only outside the *decision cone*, but also inside it.

The method has an additional advantage by rules or rule sets, in which the adornment of the recursive predicate changes during the different iteration steps. For example, the *sgc*-predicate could also be defined as follows:

r1: sgc(X,Y) :- eq(X,Y).

r2: sgc(X,Y) :- par(X1,X), sgc(Y1, X1), par(Y1,Y).

The only difference is in the body of *r2*, where the arguments of *sgc X1* and *Y1* have changed their places.

In such cases the magic-sets-transformation method (and the derived methods) requires a generation of a separate magic-set for each adornment. This may increase considerable the volume of the rewritten program. On contrary, the proposed method can be applied in such cases without any changes.

To test the proposed algorithms a software environment has been developed. It includes: a text editor for the program in declarative language, a language processor implementing the proposed algorithms and links to standard SQL-machines.

Tests have been made with InterBase Server and Oracle over many rule sets and database structures. The results, regarding speed and memory consumption, are comparable with the results of the methods based on rewriting techniques.

REFERENCES

[1] Bancilhon F.; David Maier D., Sagiv Y. & Ullman J. (1986), Magic sets and other strange ways to implement logic programs, *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pp. 1-15, ISBN: 0-89791-179-2, Cambridge, Massachusetts, United States, 1986, ACM Press New York, NY, USA.

[2] Beeri C. & Ramakrishnan R., On the Power of Magic, *Proceedings of the sixth ACM SIGACT-SIGMOD symposium on Principles of Database Systems*, pp. 269-284, ISBN: 0-89791-223-3, San Diego, California, United States, 1987, ACM Press New York, NY, USA.

[3] Ceri S.; Gottlob G. & Tanka L. (1990), *Logic Programming and Databases*, Springer-Verlag, ISBN 3-540-51728-6, Berlin.

[4] Henschen L. & Naqvi S. (1984), On compiling queries in recursive first-order databases, *Journal of the ACM (JACM)*, Vol. 31, No 1, January 1984, pp. 47-85, ISSN: 0004-5411.

[5] Iltschev V., Bottom-Up Method for Processing Nonrecursive Sets of Rules, *13-th International DAAAM Symposium,* pp. 223-224, ISBN 3-901509-20-1, 23-26 October, 2002, Vienna University of Technology, Vienna, Austria

[6] Kifer M. & Lozinskii E. (1986), Filtering Data Flow in Deductive Databases, *Proceedings of the ICDT'86*, Ausiello G., Atzeni P. (Eds.), pp. 186-202, ISBN: 3-540-17187-8, Rome, Italy, September 1986, Springer-Verlag, Berlin.

[7] Ullman J. (1985), Implementation of logical query languages for databases, *ACM Transactions on Database Systems (TODS)*, Vol. 10, No 3, September 1985, pp. 289 - 321, ISSN: 0362-5915.

ABOUT THE AUTHOR

Velko Iltchev, Department of Computer Systems, Technical University - Plovdiv, 4000 Plovdiv, Bulgaria, Mobile phone: ++359 98 410233, E-mail: v.iltchev@ieee.org.