STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVIII**, Number 1, 2003

AN APPROACH ON SEMANTIC QUERY OPTIMIZATION FOR DEDUCTIVE DATABASES

ADRIAN ONEŢ

Abstract. In this article we present a learning method to obtain rules for the semantic query optimization in deductive databases. Semantic query optimization can dramatically speed up deductive database query answering by knowledge intensive reformulation. We will present a learning method for rules that will help to semantically optimize queries for deductive databases.i We tried to change the algorithm in [2] to work for deductive database as well in this direction we propose a method for an approximate cost evaluation for deductive database predicates.

1. Introduction.

The semantic query optimization (SQO) is based on the use of the semantic rules, as it is *There is no river trough Carei city.* Using these rules, we can reformulate the queries in lower cost ones, for example: *Which are the cities that are in the Carei neighborhood?* (in the following we will consider that two cities are neighbors if there exists a river that connects these two cities). Using the semantic rules we can answer this query even without accessing the deductive database, so we will obtain a 100% cost reduction. Average savings from 20 to 40 percents are reported in the literature.

Unlike other systems, conceived for deriving rules from one database table, the Hsu&Block inductive method [2] of learning can learn semantic rules from a database with several relations (and, in our case, by using several base or derived predicates). For instance, if we consider a database that contains three relations *pupils, bachelor_grades, coordinator*, the bachelor grades can concord with the papers coordinator (e.g. All who have Mestereanu as coordinator obtained grades superior to 9 at the bachelor exam). The learning algorithm can select the relevant ways from the disjunction of two or more predicates (which is often made by the user). By using the semantic relations that describe some regularities in the disjunction of two or more predicates, our optimizer will be more than efficient

Received by the editors: March 15, 2003.
2000 *Mathematics Subject Classification.* 68U35.
1998 *CR Categories and Descriptors.* I.2.1 [**Computing Methodologies**]: Artificial Intelligence – *Appplications and Expert Systems.*

in the diminution of more complex queries execution price. We need to underline that this mechanism is profitable in a static database (without many updating in the extensional database as well as in the intensional one).

2. The semantic query optimization

The semantic query optimization is applied to different database types. Although the basic pattern of the semantic optimization refers only to the conjunctive queries, it can also be extended to other types, more complex. The general idea is that the complex queries can be decomposed in one or more conjunctive queries; after that the system can apply the semantic optimization of these queries. In this chapter we'll focus only on the conjunctive queries.

For an easier comprehension, we'll use the following knowledge database:
The extensional database structure:
Town [Name, Components]
Descriptions [Type_Component, Description]
Integration [Component, Type_Component]
Fraternity [Name_Town1, Name_Town2, Date]

with the following meaning: the relation **town** contains the name and the different components in this town (e.g. the town: *Bucharest* has the component: *Art_Museum*). The second relation, **descriptions**, obtains, for each component type a description of the respective town (e.g. if the component *Art_Museum* belongs to the component type *Historical_Objective* - according to the relation **integration** - and if in the relation description we have the tuple: Type_Component: *Historical_Objective* Description: *Tourism*, than we can say that Bucharest town is a touristic town). As we have already shown, the relation integration tells us to which type_component belongs a component by a transitivity relation. Finally, the relation **fraternity** gives us information about towns which are united, such as the date when they established the fraternity.

The intensional database structure: (in order to describe the intensional database, we maintain the Prolog syntax regarding to the variables and constants name)

objectives(Locality, Objective) :-
town(Locality, Component),
type_objective(Component, Type_component),
descriptions(Type_component, Objective).

type_objective(Component, Type_component) :-
integration(Component, Type_component).

type_objective(Component, Type_component) :-
integration(Component, Type_component1),
type_objective(Type_component1, Type_component).

We consider the following data in the extensional database:

Town

CAREI	CASTLE
CAREI	PARK
TURDA	FACTORY
CLUJ	SQUARE
CLUJ	BOTANIC_GARDEN
DEJ	FACTORY

Descriptions

HISTORICAL_OBJECTIVE	TOURISM
WORKS	POLLUTION
GREEN_SAPCES	BEAUTY
COMMERCIAL_PLACES	BUSINESS

Integration

CASTEL	MONUMENT
MONUMENT	HISTORICAL_OBJECTIVE
FACTORY	WORKS
PARK	GREEN_SPACES
SQUARE	COMMERCIAL_PLACES
BOTANIC_GARDEN	GREEN_SPACES

Fraternity

CAREI	DEJ
TURDA	CLUJ
CLUJ	DEJ

The main principle of the semantic query optimization is based on finding the equivalent queries for the initial query, but at a lower cost. The construction of the equivalent queries with the initial query is realized by using some semantic rules, which will be learned by the system from the previous queries. By a lower cost of the queries we understand a real cost approximation (the exact calculus of the cost would determine the diminution of the algorithm efficiency).

The difference between the syntactic optimization and the semantic one consists in using the semantic knowledge for expanding the search field for the semantic optimization. The conventional syntactic optimizations search the cheaper equivalent queries from the logical point of view for the initial queries [5] (the optimizations which re-sort literals/constraints belong to this category). The semantic optimization, on the other hand, searches the queries with the lowest cost equivalent to the initial query, by giving some semantic information. That is why, this type of optimization has a bigger search field and the lowest cost is also more probable comparing to the queries obtained by the syntactic optimization.

3. The learning model

In this sequence we try to present a learning model of the semantic rules (model presented in [2]). The figure 1 shows the database organization with a semantic optimizer and a learning system. The optimizer uses the semantic rules in order to optimize the queries and to send the optimized queries to rule on the deductive database in order to find the result. When the deductive database is dealing with a complex query (we mean expansive), the optimizer connects the learning system to learn a set of rules which are to be used for the optimization of other similar queries. The system will learn gradually sets of rules used for optimization.

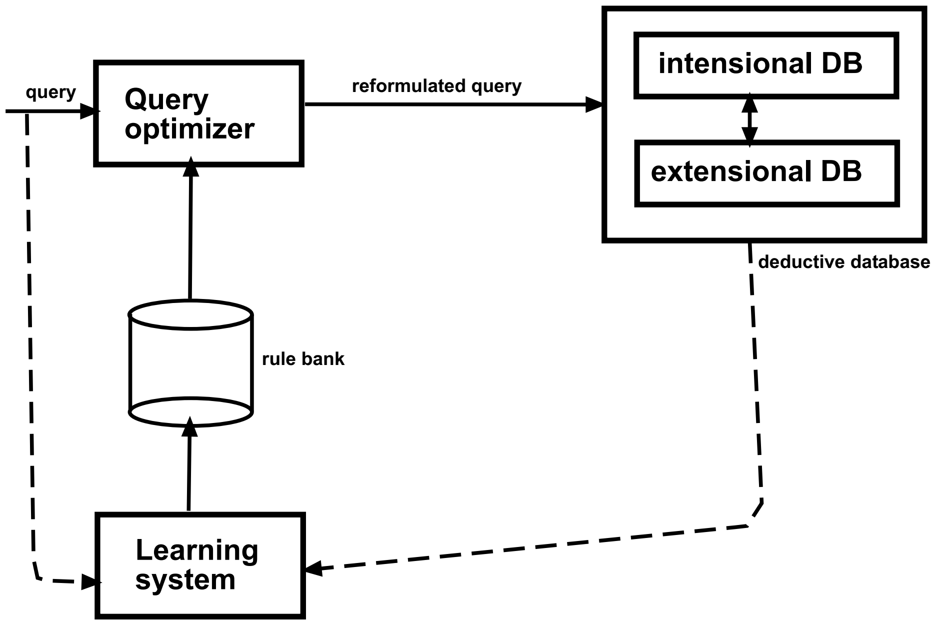

Fig.1 The structure of a knowledge database with a semantic optimizer

In the figure 2 there is a simple example of such a learning/optimization pattern. This model consists in two components [Chun95], a learning inductive component and an operational one. A query will call the learning component; afterward the system will apply an inductive learning algorithm to induct an alternative query of the initial query, but at a lower cost. Afterwards, the operationalization component, using the initial query and the alternative query learned, deduce a set of semantic rules.

In this example, the tuples from the relation are considered positive or negative depending of the query satisfaction or non satisfaction. The alternative query will have to cover only the positive instances of the relation, thus the response to the alternative query will be the same as for the main query.

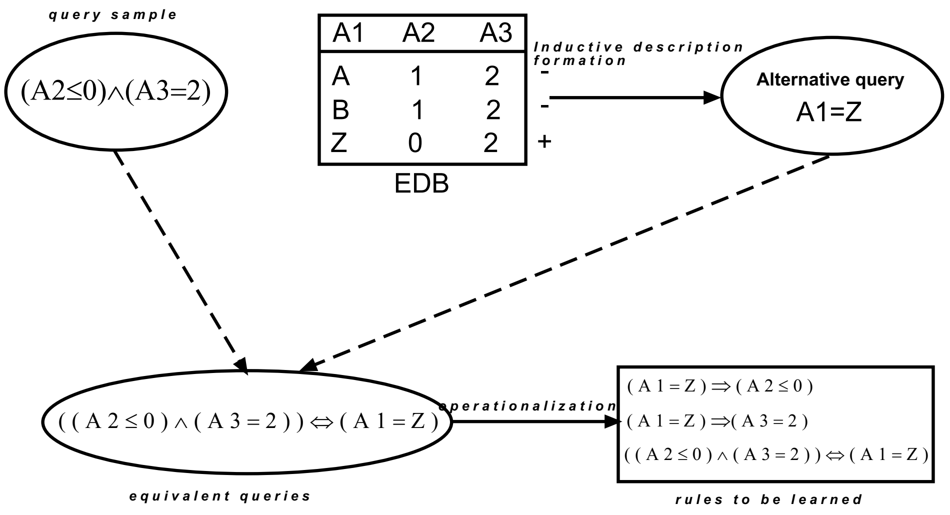

Fig.2 A learning pattern

Given a set of queries considered positive or negative, the problem of finding a description which covers only the positive instances, is named *supervised inductive learning* [2]. The more difficult problem is to compare the cost of different queries (when choosing the most appropriate semantic rule). If we use a calculus of the real cost, this optimization algorithm could not justify its objectives (the cost of such a calculus is big enough, sometimes even impossible to calculate). In this chapter we propose less expansive method which approximates the cost of such queries.

The operationalization component deduces semantic rules by using two equivalent queries. These two queries consist in two phases. In the first one, the system transforms the equivalence of the two queries into Horn clauses. For instance, given the queries: $(A2 \leq 0) \wedge (A3 = 2)$ and the equivalent query A1='Z', they will be transformed in two clauses:
1)$(A2 \leq 0) \wedge (A3 = 2) \Rightarrow A1 =' Z'$
2)$A1 =' Z' \Rightarrow (A2 \leq 0) \wedge (A3 = 2)$
The second rule can be expanded again in order to satisfy the Horn clause syntax:
3)$A1 =' Z' \Rightarrow (A2 < 0)$
4)$A1 =' Z' \Rightarrow (A3 = 2)$

After transformation, we obtain the rules 1), 3) and 4) which satisfy our syntactic demands. In the second phase, the system tries to compress the antecedents

of these rules for reducing their cost. Thus, if the rules have more than one antecedent, can be used the *greedy* minimal cover algorithm in order to eliminate the respective constraints. The minimal cover problem is to find a sub-set from a collection of sets, thus the reunion of the sub-set sets is equal to the reunion of all sets. Denying both of the first rule parts, we obtain:
$\neg(A1 =' Z') \Rightarrow \neg(A2 \leq 0) \vee \neg(A3 = 2)$

Thus, for the clause 1), we have the following problem: given a set collection which satisfy $\neg(A2 \leq 0) \vee \neg(A3 = 2)$ find the minimum number of sets which satisfy $\neg(A1 =' Z')$. If we suppose that the minimum set of the sets which covers $\neg(A1 =' Z')$ is $\neg(A2 \leq 0)$, then, in this case we can also eliminate $\neg(A3 = 2)$ from the rule and, after the clause denying, we obtain: $(A2 \leq 0) \Rightarrow A1 =' Z'$

4. Alternative query learning

In this sequence, we present an inductive learning method of the alternative queries with reduced cost. In the figure 1, it was given an example with only one predicate, but, usually, the deductive databases consist in more predicates (the basic databases as well as the derived ones), and the queries structure often implies relations of the join type. The inductive learning model described is able to learn conjunctive queries at reduced cost from the deductive databases with more predicates.

Before describing the model, we have to introduce two terms that describe the queries obtained, such as: **internal disjunctions** (constraints over one attribute value), **join constraints** (they specify a constraint over one or several attributes from different predicates) [4].

The learning model is an extension of the *greedy* algorithm which learns internal disjunctions from one database created with one table, algorithm proposed by Haussler [3]. This algorithm starts from an empty hypothesis of the concept to be learned, then continues by constructing a set of candidate constraints which respect all the positive instances, in order to choose the most promising constraints by using one heuristic function such *gain/cost*, which is added to the hypothesis. This process is repeated until negative instance doesn't satisfy the hypothesis.

The algorithm has as entrance the Q query and the predicates from the deductive database. We call the **base relation** the relation that must be accessed for the initial queries. If the output attributes of the query are connected to different predicates, then the base relation is the relation resulted from the join of these relations. For example, if, in our case, we consider the query *What towns have as main objective the beauty* the base relation would be given by the predicate **objectives**, the tuples of this relation (the ones marked by + are those who satisfy the query):

Objectives

CAREI	TOURISM	
CAREI	BEAUTY	+
TURDA	POLLUTION	
CLUJ	BUSINESS	
CLUJ	BEAUTY	+
DEJ	POLLUTION	

Initially, the system determines the base relation of the entrance query, then it marks its instances as being positive or negative (an instance will be marked as positive if it satisfies the query, and as negative on the contrary)

```
Algorithm Learning alternative queries
Input: Q- entrance query, DDB - deductive database
Output: AQ - alternative query
Let r = base relation for Q
Let AQ = ∅ - alternative query
Let C = ∅ - candidate constraints set
  It builds candidate constraints for r and added to C
Repeat
      Evaluates the rapport gain/cost of the constraints from C
      Let c = the constraint with the lowest value for gain/cost from C
      if gain(c)>0 then
        add c to AQ
        C = C - c
        if AQ⇔Q then return AQ
        else
          if c is a constraint of the join type on the new relation r' then
            build candidate constraints for r' and it is added to C
          end if
        end if
      end if
until gain(c) = 0
  return failure - because it wasn't found AQ equivalent to Q
end algorithm
```

In this algorithm we still have to explain how the constraints were built and how the calculus of the heuristic function *gain/cost* was made.

5. Building the candidate constraints

For every attribute from the base relation, the system can build an internal disjunction, as well as a candidate constraint, by generalizing the positive instances attributes value. For the query given as example for the first attribute from the base relation we have the following values: *Carei*, *Cluj*, and, if we consider the second attribute, we have the value *beauty.* Similarly, the system can also consider the constraints of join type as well as the candidate constraints if it consists in all the positive instances. If, for example, we have join between **objectives** and **fraternity**, we'll obtain the relation (we marked with + the instances which satisfy the query)

Objectives$\otimes$**Fraternity**

CAREI	TOURISM	CAREI	DEJ	
CAREI	BEAUTY	CAREI	DEJ	+
TURDA	POLLUTION	TURDA	CLUJ	
CLUJ	BUSINESS	CLUJ	DEJ	
CLUJ	BEAUTY	CLUJ	DEJ	+

We can notice that this kind of relation can help us finding a candidate constraint, for example for the second attribute of the predicate fraternity, we have, for all the positive instances the value DEJ (we have to consider, of course, the cost as well, but in most of the cases, the cost of one join is lower then the cost of traversal of one relation of the join [6]).

6. The evaluation of the candidate constraints

Once built the set of the candidate constraints, we will have to establish which is the most promising one and to add it to the hypothesis (see the algorithm). To do it, we'll have to evaluate the *gain/cost* value for each constraint, where by *gain* we mean the number of excluded negative instances. For the basic predicates there is one set of algorithms that approximate the cost function of their physical structure [5],[6]); in the case of derived predicates, their cost estimation is more difficult. We'll present as follows an estimation method for the cost of these predicates (the costs also depend, of course, on the basic predicates access).

The cost estimation method is a rewriting method. Consequently, we add a new basic predicate **estimated_cost(predicate, cost)** whose argument is the predicate, that is the estimated cost of the given predicate. The algorithm consists in the modification of the tuples corresponding to the relation function of the introduced relations (thus, as it is no longer necessary to access these predicates when calculating, but only to consult the relation corresponding to the predicate estimated_cost). Initially, for each derived predicate, will exist a corresponding tuple with the initial value is 1. For our example, we will have:

estimated_cost

objectives	1
type_objective	1

The next step consists in rewriting the rules so that they can evaluate the approximated cost of a predicate. To do this, we'll have to add another attribute to every derived predicate and that will represent the cost. Thus, this cost will increase at every call of the clause function of the dimensions of the relations corresponding to the other predicates and function of the dimension of the relations corresponding to the basic predicates (here we can optimize by introducing other parameters such as indexes [5]) which compose the respective clause. Thus, for our example, we'll have the following intensional database (after rewriting):

objectives(Locality, Objective, Cost_objectives) :-
town(Locality, Component),
type_objective(Component, Type_component, Cost_type_objective),
descriptions(Type_component, Objective).
Cost_objectives=dim_town * Cost_type_objective * dim_description.

type_objective(Component, Type_Component, Cost_type_objective) :-
integration(Component, Type_Component),
Cost_type_objective=dim_integration.

type_objective(Component, Type_Component, Cost_type_objective) :-
integration(Component, Type_Component1),
type_objective(Type_Component1, Type_Component,
Cost_type_objective_intermediary),
Cost_type_objective = dim_integration *Cost_type_objective_intermediary.

Where dim_town, dim_description, dim_integration are some variables representing the cost of the access to the relations: Objective, Description, and Integration. All that remains to do is to memorize the results in the relation corresponding to the predicate **estimated_cost**, which can be realized by using an update predicate (by updating one predicate value cost we'll understand the change with the cost biggest value) of the tuples at the call of each predicate. This is not recommended because there can exist internal predicates which are not called explicitly by the query, but called only by other predicates whose cost is not evaluated. That is why, we have chosen (for now - in the future, we are counting on finding a better solution) to integrate this update predicate in every clause, corresponding to the heading predicate. Thus, for the previous example, we have (supposing that the update predicate is named **cost_update(predicate, cost)**):

objectives(Locality, Objective, Cost_objectives) :-
town(Locality, Component),
type_objective(Component, Type_Component, Cost_type_objective),
description(Type_Component, Objective),
Cost_objectives=dim_town * Cost_type_objective * dim_description,
Cost_update(objectives, Cost_objectives).

type_objective(Component, Type_Component, Cost_type_objective) :-

integration(Component,Type_Component),
Cost_type_objective= dim_integration,
Cost_update(type_objective, Cost_type_objective).

type_objective(Component, Type_Component, Cost_type_objective) :-
integration(Component,Type_Component1),
type_objective(Type_Component1, Type_Component, Cost_type_objective_inter),
Cost_type_objective = dim_integration * cost_type_objective_inter,
Cost_update(type_objective, Cost_type_objective).

The predicate **Cost_update**/2, as we have specified, will change the predicate (the first parameter) cost value (the second parameter) in the relation corresponding to the predicate **estimated_cost**/2, only if the cost value given by the second parameter is bigger then the value found in the relation.

This algorithm works very good for the systems whose evaluation is bottom-up (it ignores the parameters link); in the top-down case, there are differences between the cost of a predicate which has no linked parameter and the cost of a predicate which has all the parameters linked. That is why, for the top-down evaluation systems, we change the previous algorithm so that this evaluation still makes the difference between the parameters different ways of linking. The modification consists in adding a new attribute to the relation **estimated_cost**/3 that tells us for what link type the cost is calculated. Consequently, in our example, the relation contains initially:

estimated_cost

objectives	bb	1
objectives	bf	1
objectives	fb	1
objectives	ff	1
type_objective	bb	1
type_objective	bf	1
type_objective	fb	1
type_objective	ff	1

We can notice that we only considered the parameters that entered in the initial relation. The clauses will be also modified, meaning that we will add one parameter from the list type to each derived predicate. This parameter will tell us about the way that the link is realised for the predicate, at that moment; thus, the way of a variable link also depends on the way that the previous predicate was evaluated. Our intensional base will be as follows:

objectives(Locality, Objective, Cost_objectives, Link_Objectives) :-
town(Locality, Component),
type_objective(Component, Type_Component, Cost_type_objective, [b,f]),
descriptions(Type_Component, Objective),

Cost_objectives=dim_town * Cost_type_objective * dim_descriptions,
Cost_update(objectives, Cost_objectives, Link_Objectives).

type_objective(Component, Type_Component, Cost_type_obj, Link_type_obj) :-
integrare(Component, Type_Component),
Cost_type_obj= dim_integrare,
Cost_update(type_objective, Cost_type_obj, Link_type_obj).

type_objective(Component, Type_Component, Cost_type_obj, Link_type_obj):-
integration(Component,Type_Component1),
value(2,Link_type_obj,L),
type_objective(Type_Component1,Type_Component,Cost_type_obj_inter,[b,L]),
Cost_type_obj = dim_integration*cost_type_obj_inter,
Cost_update(type_obj, Cost_type_obj,Link_type_obj).

We used the **value**/3 predicate that indicates the way of link of the parameter given by number by the first argument from the link list given as the second argument, and the third argument will make us return to the respective value. For example, the predicate **value**/3 description will be:
value(1,[A|_],A).
value(N,[_|T],A):- *succ*(N1,N), *value*(N1,T,A).

At every derived predicate call, we'll also have to precise the way of parameters linking (for instance, the query *What are the towns that have as principal objective beauty* is described as follows: ?:- objectives(Town, beauty, Cost, [f,b])).

We notice that, in this case, the **cost_update** predicate has three arguments (was also added the link list - a tuple from **estimated_cost** is once identified by the predicate name and by the way of the parameters linking).

In this algorithm constants were used for the cost of the relations corresponding to the main predicates, these constants are calculated according to the formula described by [6].

After this modification of the database, the cost calculus for the candidate constraints is no longer an issue, because we have already calculated these costs in the **estimated_cost** relation; thus, using one traversal of this relation, we can determine every derived predicate that belongs to a candidate constraint.

7. The search in the candidate constraints domain

When a join type constraint is chosen, a new relation is introduced in the candidate constraints domain. The system can choose to add new constraints on the account of the new relation from the hypothesis or to consider the attributes of the old relation. When a join type constraint is added, the domain is devised in two levels, if to the new introduced relation constraints a new join type constraint is added; then, the domain will be devised in three levels and so on. The exhaustive evaluation of the gain/cost for all the candidate constraints is not at all practical when we deal with a big and complex database. The system introduce by [2] prefers the model that favors the candidate constraints of the new introduced relation. Thus, when a join type constraint is selected, the system will consider only those candidate constraints from the new level, until the system builds a hypothesis that covers all the positive instances and no negative instance (that means that it achieved its purpose), or until it can not find other constraints that have a positive cost on that level. In the last case, the system returns to the previous level, continuing searching. This evaluation method was chosen because a join type constraint is less probable chosen from the rest of the constraints (usually, the join constraints have a bigger cost); if this constraint was chosen, it means that the rest of the constraints have a very high cost or a negative gain.

8. Conclusion

In this article we have shown that the knowledge required for semantic query optimization can be learned under the guidance of the input queries. We have described a method to approximate the query execution cost in deductive databases. We can use this method in the inductive learning algorithm described by [2] for semantic query optimization in deductive databases. A limitation of this semantic query optimization approach is that there is no mechanism to deal with changes in the deductive database. This problem can be solved as follows: when the deductive database is changed, a maintenance system will be used to update the rule bank so that there will remain only those rules which respects the updates made to the deductive database.

References

[1] Weidong Chen : "Query Evaluation in Deductive Databases with Alternating Fixpoint Semantics", ACM Transactions on Database Systems vol. 20/3, 1995, pp. 239-287.

[2] Chun-San Hsu, Craig A. Knoblock: "Using inductive learning to generate rules for semantic query optimization", Advances in Knowledge Discovery and Data Mining, 1996, pp. 425-445.

[3] Haussler,D:"Quantifying Inductive Bias: AI Learning Algrithms and Valiant's Learning Framework.",Artificial Intelligence 36, 1988, pp. 177-221.

[4] Raghu Ramakrishnan:"Database Management Systems", WCB McGraw Hill, 1998.

[5] Jeffrey D. Ullman: "Principles of Databases and Knowledge-Base Systems, Vol I: Classical Database Systems", Computer Science Press, New York, 1988.

[6] Jeffrey D. Ullman: "Principles of Databases and Knowledge-Base Systems, Vol II: The new technologies", Computer Science Press, New York, 1989.

[7] S. Ceri, G. Gottlob, L. Tanca: "Logic Programming and Databases", Springer-Verlag Berlin Heidelberg 1990.

[8] L. Warshaw, Daniel P. Miranker: "Rule-Based Query Optimization, Revisited",CIKM, 1999, pp. 267-275

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

E-mail address: `adrian@cs.ubbcluj.ro`