Metapredicate Optimization for Datalog Queries through Program Analysis

F. Bueno
DIA
Madrid, Spain
Polytechnic University of Madrid (UPM)
bueno@fi.upm.es

J. Correas
DSIC
Madrid, Spain
Complutense University of Madrid (UCM)
jcorreas@fdi.ucm.es

F. Sáenz-Pérez
DISIA
Madrid, Spain
Complutense University of Madrid (UCM)
fernan@sip.ucm.es

Some systems extend Datalog in order to allow the use of constructions in which several queries are composed to produce the set of resulting tuples. These constructions include outer joins, aggregate and grouping predicates, as well as, to some extent, negation. Typically, the result of such constructions depends on the subset of the tuples in the sets initially computed. In order to optimize for efficiency these compound queries, it would be interesting to determine in advance the subsets involved in the compound construct. Static analysis can be used at compile-time to infer an over-approximation of such subsets. Very precise abstract interpretation-based static analyzers have been developed for logic languages, and in particular the use of type domains allow to infer descriptive types for the arguments of a given predicate. Using the extensional description of the types inferred, the Datalog program can then be transformed to use the inferred subsets instead of the original queries. Here, we propose a source-to-source transformation of Datalog programs based on static analysis for optimizing queries involving outer join, negation, aggregate and grouping predicates. This approach has been tested in the DES system, using CiaoPP (a language preprocessor for Prolog) for inferring descriptive types. Some preliminary experiments show promising results.

1 Introduction

Several deductive database systems have emerged along time, mostly born from academic efforts. See, for instance DLV [17], XSB [27], bddbddb [16], LDL++ [2], DES [24], ConceptBase [15], and .QL [20]. Their deductive engines are based mainly on two approaches: magic sets transformations [4] and memoization (tabling) [10, 29]. With the former approach, logic programs are rewritten to implement a bottom-up (forward-chaining) efficient computation, enabling efficient methods for doing massive joins (e.g., [6]). With the latter one, a lookup table is used to avoid recomputations, therefore saving computation time (e.g., [21]).

Although systems implementing tabling are subject of further optimizations (e.g., indexing using tries in XSB), the problem of handling large amounts of data still remains. In fact, lookup tables come at the cost of memory consumption. In this work, we focus on how to enhance the implementation of a deductive database engine based on tabling by reducing the lookup table contents related to the computation of several primitives. By using abstract interpretation techniques as a static analysis for Datalog programs, we gather type information to prune computations (and, therefore, lookup table contents) of some Datalog primitives.

Such primitives are the metapredicates for outer joins, negation, and aggregation [25, 26]. Any of these primitives includes other predicates as arguments, so that the result of a metapredicate construction depends on a subset of the tuples in the set that represents the whole meaning of each predicate it takes as an argument. Handling a metapredicate like those typically requires program stratification [30], which

Contribution to:
PROLE 2016

isolates the caller from the callee in two different strata. Computing by stratum implies losing important information about the amount of work which is really needed to perform at run-time. In particular, the possible instantiations a goal can take are lost in general. This is to say that the precious saving work that tabling does is partially lost when segmenting the program into strata. Thus, our goal in this work is trying to recover as much information as possible by taking advantage of static analysis, which can narrow the search space of open queries resulting from stratification. Each of these open queries represent the predicate a metapredicate can take as an argument.

Here, we use two current systems for validating our proposal: CiaoPP [14] as a state-of-the-art abstract interpretation-based processor for logic programs, and DES [24] as a current deductive database system embodying special table-based primitives which are quite suited to optimizations. We perform some experiments that show the effectiveness of this proposal.

This paper is organized as follows. Section 2 introduces the metapredicates subject to optimizations. Section 3 describes the implementation details of the DES system to allow the reader to understand the decisions geared towards its optimization. Section 4 includes an overview of the type inference system in CiaoPP and the information we gather from its analysis, which is applied to source-to-source transformations with the goal to reduce run-time work. Preliminary experiments are presented and analyzed in Section 5. Finally, some conclusions and future work are summarized in Section 6.

2 Metapredicates Subject to Optimizations

We focus on some metapredicates in Datalog which represent compound goals, such that their solving requires computing the sets of tuples of some argument relations (predicates) and then composing the results. The composed results will usually come from only a subset of the tuples in the sets initially computed. Each of these subsets is built from the goal instantiation the predicate call can take. Our aim is to determine beforehand such subsets as precisely as possible in order to reduce the overhead of the computation of the compound goal.

2.1 Negation

A negated goal in Datalog must be non-compound (simple). Negation amounts to checking whether the tuple corresponding to such simple goals belongs to the corresponding relation or not. In DES, negation is represented by the construction `not(G)`, where `G` is a simple goal, which must be ground upon execution. Computing negation follows a program stratification that classifies predicates into strata, guided by negated goals. The goal `G` has to be computed before its negation. So, DES computes the set of tuples in the relation of `G` and then checks if the ground atom belongs to it. If we can statically obtain sets of possible values for the arguments of goal `G`, then the computation of the set of tuples could be reduced to only those with arguments in the given sets. Furthermore, if we knew the exact values of the goal during execution, then the computed sets of tuples would be reduced to a singleton or an empty set (depending on whether the ground atom of the goal belongs to the relation or not). In an extreme case, whether such set is empty or not could even be determined beforehand, depending on the quality of the information at hand.

Example 2.1. Let `not(p(X,Y))` be a goal. DES computes the sets of tuples for relation `p`/2 and then checks whether the ground goal `p(X,Y)`θ is in that set, where θ is the substitution built along solving that grounds the goal. Instead, if we knew that there are sets of values S_x and S_y such that, upon execution, $X \in S_x$ and $Y \in S_y$, we could compute only the set of tuples of relation `p`/2 such that its first argument

is in S_x and its second argument in S_y. If S_x and S_y were singleton sets, we would compute a set with at most one value (it would be empty if the tuple does not belong to the relation). Moreover, if we knew beforehand whether atom(s) `p(X,Y)` with $X \in S_x$ and $Y \in S_y$ either are or are not in the success set of p/2, then the execution of the negated goal will simply amount to either failing or succeeding, straightforwardly avoiding the computation of any set of tuples.

2.2 Outer Joins

Outer joins are usual constructs in relational databases [30]. An outer join includes in its result the tuples in the Cartesian product of a relation A and a relation B that satisfy a third relation C. Satisfying a relation corresponds to check whether its semantics is non-empty. In addition, the result is enlarged with the tuples in one of the relations (A or B) which do not have a counterpart in the other relation (B or A, respectively), and the values corresponding to the relation with no corresponding tuple are set to `null`. If this is true for relation A in the Cartesian product $A \times B$ then it is a left outer join; if it is true for B then it is a right outer join; if it is true for both then it is a full outer join. Here, and as commonplace, "left" ("right") means that all the tuples to the left (right, respectively) of the outer join operator must be included in the result.

In DES, the left (right and full, respectively) outer join [25] corresponds to the construction `lj(A,B,C)` (`rj(A,B,C)` and `fj(A,B,C)`, respectively), with `A`, `B`, and `C` simple goals. It represents the tuples in the Cartesian product `A`×`B` which satisfy `C`. Similar to the negation case, relations `A` and `B` are set in a lower stratum than the predicate for the rule in which the outer joins occurs. So, DES computes the sets of tuples corresponding to the relations of `A` *and* `B`, then the tuples in the Cartesian product, and then reduces this set by filtering only the tuples that satisfy `C` (depending on the outer join, the result is enlarged with the tuples of one or both relations that do not satisfy `C`, with the absent arguments being filled with a `null` value). However, in a particular program execution, the arguments of either `A` *or* `B` might have given values, and the goal `C` also restricts the possible values of arguments of the tuples in the overall result. If we knew beforehand such sets of values, the computation of the tuples for `A` *and* `B` could be reduced to only those that satisfy them.

Example 2.2. Let `lj(p(X,Y),q(A,B),A=Y)` be a goal. DES computes the sets of tuples for relations p/2 and q/2, obtains the Cartesian product, and then filters only those that satisfy that the first argument of q/2 is equal to the second one of p/2. Instead, if we knew that there is a set of values S_x such that, upon execution, $X \in S_x$, we could compute only the set of tuples of relation p/2 such that its first argument is in S_x. Similarly, this can be done also for relation q/2. Moreover, if we knew a set S_y such that a tuple is in the computed set for p/2 if and only if its second argument is in S_y, then we could compute only the set of tuples of relation q/2 such that its first argument is in S_y (since `A=Y` must hold).

2.3 Aggregate Predicates

We restrict our attention here to classical aggregation operations in relational theory. In particular to minimum (`min`), maximum (`max`), sum (`sum`), average (`avg`), and count (`count`). They compute a single value from the aggregation of the tuples in a given relation. For example, counting in DES is represented by the construction `count(G,X,C)`, where `G` is a goal, `X` a variable in an argument position of `G`, and `M` a numeric term. It is true if and only if `C` is the number of tuples such that `X` is not null in the meaning of `G`. DES computes the set of tuples in the relation of `G`, and then computes the count. Again, if we knew current sets of values for the arguments of `G` we could restrict the computation of its relation to only the tuples satisfying such values.

Example 2.3. As an example showing an application of aggregate predicates, let us consider the goal for `count` below, on the relation `employee/3` which is defined as follows:

```
              % Relation employee(Name,Department,Salary)

    employee(anderson,accounting,1200).   employee(sanders,sales,null).
    employee(andrews,accounting,1200).    employee(silver,sales,1000).
    employee(arlingon,accounting,1000).   employee(smith,sales,1000).
    employee(nolan,null,null).            employee(steel,sales,1020).
    employee(norton,null,null).           employee(sullivan,sales,null).
    employee(randall,resources,800).

DES> count(employee(N,D,S),S,T)
Info: Processing:
  answer(T) :- count(employee(N,D,S),S,[],T).
{ answer(7) }
Info: 1 tuple computed.
```

Note that the goal has been processed as an autoview [24], i.e., a rule (which is asserted to the current database) with the head predicate name `answer`, relevant variables in the goal as head arguments, and its body beign the goal. Solving an autoview amounts to call the goal `answer(C)` and eventually to remove the rule. In this case, the metapredicate `count` includes set variables (existentially quantified, similar to Prolog's `findall`) which are not relevant for the outcome (`N`, `D`, and `S`). This way, outcomes remain closed (because non-ground tuples as answers are disallowed[1]). Also note that an extra argument occurs in the rule, which corresponds to the list of variables for grouping (more on this on the next subsection). Here, no grouping is specified and the list appears empty. The number `7` in the outcome reflects the number of tuples in `employee` such that no null occurs in the salary.

DES computes the relation independently of the values that, upon execution, variables may take. Therefore, knowing in advance what possible calls are allowed to each aggregate goal would lead to smaller relations to work on.

2.4 Aggregate Predicates and Grouping

Following (extended) relational theory, another useful operation is the grouping predicate (`group by`). This predicate builds groups (subsets) of a given relation so that tuples with the same values for each attribute of a given list of attributes belong to a group. The first form of grouping that DES supports is in aggregate predicates, which are easily achieved in the context of a clause. The variables of the relation in the aggregate predicate occurring in the clause head are the arguments on which to build groups. As before, DES computes the set of tuples in the relation of `G`, so that knowing in advance its possible candidates would restrict the computation.

Example 2.4. As an example showing an application of this form of grouping, consider the following query on relation `employee/3` of Example 2.3. Grouping is applied to variable `D`, which is an argument of the tuple the query defines, which occurs also as an argument of the `employee` relation in the goal of `count`.

[1] By contrast, some constraint deductive databases include non-ground outcomes [22].

```
DES> emp_by_dep(D,C):-count(employee(N,D,S),S,C)
Info: Processing:
  emp_by_dep(D,C) :- count(employee(N,D,S),S,[D],C).
{ emp_by_dep(accounting,3),
  emp_by_dep(null,0),
  emp_by_dep(resources,1),
  emp_by_dep(sales,3) }
Info: 4 tuples computed.
```

Note that a temporary view is explicitly used here by the user by adding a head to the query, with the intention of representing the outcome (answer relation) of interest. Since the variable `D` is added to the head, it has been identified as a grouping argument and this way it is included in the grouping list of the executed temporary view. Similar to the previous example, only the tuples with non-null values in the salary are counted.

As before, DES first completely computes the relation `employee/3`, so that knowing in advance the possible allowed calls would restrict the computation indeed.

2.5 Grouping Predicate and Aggregate Functions

The other form of grouping allowed in DES is via the predicate `group_by(G,Vs,E)`, which encloses a goal `G` for which a given list of variables `Vs` builds answer sets (groups) for all possible values of these variables. For each group, the expression `E` is computed. In general, this expression contains *aggregate functions*, in contrast to the aggregate predicates aforementioned. Aggregate functions include `count/0`, `count/1`, `sum/1`, `avg/1`, `min/1`, and `max/1`. When providing an argument (a variable denoting an argument of the relation, as the variable `S` for the relation `employee(N,D,S)`) to `count`, the computed result of this function is the number of those tuples in the group so that each tuple has a non-null argument. Otherwise, all tuples are counted.

Example 2.5. Given the relation `employee/3` in Example 2.3, the following query shows an application of this predicate:

```
DES> group_by(employee(N,D,S),[D],R=count)
Info: Processing:
  answer(D,R) :- group_by(employee(N,D,S),[D],R = count).
{ answer(accounting,3),
  answer(null,2),
  answer(resources,1),
  answer(sales,5) }
Info: 4 tuples computed.
```

As in Example 2.3, the goal has been processed as an autoview by automatically adding the head `answer(D,R)` to a clause containing the goal for `group_by` as its body. This head contains the grouping variables and the free variables in the expression `R=count`. This way, outcomes remain closed (non-ground tuples as answers are discouraged in non-constraint deductive databases). Also, the very same reasons motivated in the previous section do apply in this case for the optimization of the query.

3 Solving Metapredicates

This section briefly recalls some implementation details of the DES system to understand the decisions geared towards the optimization of the just introduced primitive metapredicates. First, it sketches tabling as implemented in DES, then recalls the notion of dependency graph and stratification, and finally what are the source-to-source transformations which are performed on such primitives.

3.1 Tabling

The computation model of DES follows a top-down-driven bottom-up fixpoint computation with tabling [26]. DES uses an extension table which stores answers to goals previously computed, as well as their calls. For the ease of the introduction, we assume an answer table and a call table to store answers and calls, respectively. Answers may be positive or negative, that is, if a call to a positive goal `p` succeeds, then, the fact `p` is added as an answer to the answer table; if a negated goal `not(p)` succeeds, then the fact `not(p)` is added. A negated goal is proved by means of negation as failure (closed world assumption). Calls are also added to the call table whenever they are solved. This allows to detect whether a call has been previously solved and the computed results in the extension table (if any) can be used.

3.2 Dependency Graph and Stratification: Negation, Outer Joins, and Aggregates

Each time a program is consulted or modified (i.e., via submitting a temporary view or changing the database), a predicate dependency graph is built [32]. This graph shows the dependencies between predicates in the program. A predicate p positively depends on q if p is in a predicate head and q is in its body. The dependency becomes negative if q is an argument of either a negation, or an outer join goal or an aggregate goal [26]. Dependencies are transitive.

This dependency graph is useful for finding a stratification for the program [32]. A stratification collects predicates into numbered strata ($1 \ldots N$) such that a predicate p which negatively depends on p must be placed in a strata higher than the strata assigned to q.

Usually, the number of strata is kept to a minimum (unless other optimizations are considered [19]). A basic bottom-up computation would solve all of the predicates of the program assigned to stratum 1, then 2, and so on, until the meaning of the whole program is found. However, DES restricts the computation to the predicates on which the query depends on. Thus, it only resorts to computing by stratum when a negative dependency occurs in the predicate dependency graph restricted to the query. Anyway, each predicate that is actually needed is solved by means of the extension table mechanism sketched in the previous section. As a consequence, many computations are avoided with respect to a naïve bottom-up implementation.

Predicates of the relational arguments of negation, outer joins and aggregate goals are collected into deeper strata in order to have their answer sets completely defined. This way, non-monotonic computations are avoided (i.e., floundering [9]).

3.3 Source-to-Source Transformations

This section explains the source-to-source transformations which are conducted during the preprocessing phase in DES. Transformations are needed to allow the user to write more readable code that cannot be solved directly. Such preprocessing is applied to outer joins, negation, and aggregate predicates.

3.3.1 Outer Joins

An outer join *oj*(A,B,C) (*oj* ∈ {lj,rj,fj}) is transformed into an equivalent (w.r.t. semantics) set of rules that the underlying engine is able to process. The transformation is guided by the need to have already computed the outer joins before trying to compute other goals in its context which are descendant in the usual SLDNF left-to-right order. Stratification allows to implement this idea in a very same way as negation is computed. Then, to enable strata classifications for outer joins, a new predicate $p*i* is introduced as an argument of the built-in, void predicate *oj*/1, which does nothing but add a negative dependency. The predicate $p*i* is to be set in a deeper strata than the predicate of the rule in which it occurs. Its arity is $|vars(A) \cup vars(B) \cup vars(C)|$. All of the facts in its meaning come from two sources: the facts in A joined with those of B which meet C, and the facts in A joined with nulls which *do not* meet C. The source-to-source transformation in this case can be seen with the example rule `v(X,Y) :- lj(s(X,U),t(V,Y),U>V)`:

```
v(X,Y) :-
    lj('$p2'(X,U,V,Y)).
'$p2'(A,B,'$NULL'(C),'$NULL'(D)) :-
    s(A,B), not('$p3'(A,B,E,F)).
'$p2'(A,B,C,D) :-
    '$p3'(A,B,C,D).
'$p3'(A,B,C,D) :-
    s(A,B), t(C,D), B > C.
```

Note that the predicate $p3 is used to compute both sources of facts, whether provided by the positive case (a *straight* call to $p3 from the second rule $p2) or the negative one (a *negated* call to $p3 in the first rule of $p2). Also note that the first rule builds the null values for the arguments of the right relation B for which no tuples are found meeting C. These null values are internally represented with the term `'$NULL'(V)`, where V is interpreted as a distinguishable, unique variable in the program signature. These null values are rendered to the user as the constant `null`. Finally, the predicate $p3 contains the (possible) hard work to be computed since it contains the Cartesian product of two tables; so it becomes a clear candidate for pruning computations when enough large amounts of data are considered. Transformations for the other outer joins are similar.

3.3.2 Aggregate Predicates and Negation

The primitives for aggregate predicates contain an additional argument for specifying the list of grouping arguments. These primitives are not allowed to be used by the user; instead, they appear as the result of a preprocessing. Recall Example 2.4, where one can notice that the submitted autoview has been translated in order to include the extra argument `[D]` denoting the list of grouping arguments. Also, a new predicate and call are built for a compound goal as an argument of a negation, as it is required to be non-compound. Consider, for instance, a disjunction as such an argument for the negation `not(p;q)`. This goal is transformed into `not('$p0')` and the following rules are added to the program:

```
'$p0' :- p.
'$p0' :- q.
```

4 Compile-Time Information to Prune Computations

Our proposal consists in taking advantage of program analysis techniques to improve the evaluation of DES programs. In our approach, we use program analysis based on Prolog semantics. We first instrument the program with Prolog predicates that mimic the semantics of the DES compound goals. The program is then analyzed and the information inferred is used to transform the program. The analysis used infers the types of calls and successes of goals in the Prolog execution. The program transformation uses analysis results to avoid tuples in the relations which are deemed irrelevant for the computation of the answer sets by the program analysis. Thus, DES program evaluation is guided towards tuples that "have a chance" of resulting in computed answers for the compound goals.

In the rest of the section we explain first the way the type information can help in program evaluation, then the type analysis we use, and finally the program transformations performed in our approach.

4.1 Static Analysis

We perform a static analysis of the program with a Prolog semantics. This gives us call and success patterns for the program goals. This information is passed on to the DES compiler, which performs the related optimizations for execution time. The analysis used is a type analysis, which gives types (i.e., sets of –possible– values) for the program variables. Let us see it in more detail:

1. Outer join. Let `lj(p(X,Y),q(A,B),A=Y)` be a program goal. Assume that `X` and `B` will be ground upon execution, `Y` and `A` will be free. Type analysis will obtain a call pattern for the goal with possible values of `X` and `B`, and a success pattern for the goal with possible values of `Y` and `A`, all of which can be used by the DES compiler to reduce the computation of the sets of tuples of relations p/2 and q/2 for the above goal.
2. Negation. Let `not(p(X,Y))` be a program goal. Note that `X` and `Y` must be ground upon execution. Type analysis will obtain a call pattern for the goal with possible values of `X` and `Y`, which can be used by the DES compiler to reduce the computation of the sets of tuples of relation p/2. A safe type analysis cannot determine that the goal will definitely succeed; however, it may determine that the goal will definitely fail. In such case, the goal can be replaced at compile time by a failure.
3. Aggregation. Let `min(p(X,Y),X,M)` be a program goal. Assume that `Y` will be ground upon execution. Type analysis will obtain a call pattern for the goal with possible values of `Y`, and also a success pattern for the goal `p(X,Y)`, with possible values of `X` that satisfy `p(X,Y)`. This information can be used by the DES compiler to reduce the computation of the sets of tuples of p/2.

4.2 Getting Information: The Ciaopp Preprocessor

We use a Prolog program analyzer already implemented in CiaoPP to infer relevant information on DES programs. CiaoPP [14] is the program preprocessor of the Ciao system [13]; it incorporates several program optimizations and various program analyses. For our purposes, the relevant analysis is that of types. This analysis is performed by a fixpoint computation based on abstract interpretation, in an abstract domain of *regular types*. Regular types [7] are well suited for describing terms in the computation of logic programs. They also correspond to *regular programs* so that a regular type can be described by one such program. For example, the type of lists of numbers in Prolog will be described by the following program:

```
list([]).
list([X|Xs]) :- num(X), list(Xs).
```

The CiaoPP system outputs the inferred types in such a form, together with an adorned program which asserts at which program points the types hold. The analysis infers types for the call and the success of goals in the program. Analysis results can be given at the predicate level, using assertions that state that some properties hold for any call to a given predicate. More interestingly for this paper, analysis information can also be given, in the form of a call to a predicate named `true` with one argument, to specify information at particular program points. For example, following the previous example for a Prolog program,

```
append([X|Xs],Ys,[X|Zs]) :- true(list(Ys)), append(Xs,Ys,Zs).
```

means that, during the computation of the program, it is always the case that the recursive call to `append` is performed with a list in its second argument.

The analysis is able to infer such *descriptive* types: types defining the form of terms constructed in the computation of the program. (As opposed to prescriptive types, that prescribe the terms the program is allowed to compute.) The abstract domain used is based on regular grammars that describe the regular types which can be constructed with the constants and functors of the program under analysis plus a small set of predefined types. The predefined types account for particular interesting cases which are not regular, such as numbers, integers, characters, and the like.

4.3 Program Transformation

The program transformation proposed in this paper consists of the following steps:

1. The program is transformed to avoid the use of DES-specific built-in predicates. Such predicates are replaced by simple, "fake" predicates that preserve the semantics with respect to type information. For example, instead of using the actual implementation of `min/3` (minimum value of a relation argument) in DES, a single fact `min(_,X,X)` is used to propagate the type information between second (relation argument) and third (minimum) arguments of `min`. The rest of built-ins are defined similarly.
2. These simple definitions are unfolded into the given program.
3. The resulting program is analysed for types, and the inferred program point information is added to the program.
4. The original program is transformed, replacing the goals used in metapredicates with specialized versions generated using the program point information gathered in the previous step.

We illustrate the program transformation by means of a running example. Let us consider the program below. Its single entry point is the predicate `main/1`, which indexes on three clauses, one for each case we are considering: outer join, negation, and aggregation.

```
:- module(examples,[main/1]).
:- use_module(des).

main(lj)  :- lj_t_on_s(1,_,_).
main(not) :- not_l(1), not_l(10).
main(min) :- min_t_on_first(1,_).
```

```
lj_t_on_s(X,Y,V) :- lj( t(X,Y), s(U,V), X<U ).

not_l(X) :- not( l(X) ).

min_t_on_first(X,M) :- min( t(X,Y), X, M ).

s(1,2). s(1,3). s(2,2). s(3,2). s(4,2). s(5,2). s(6,2).
t(1,1). t(1,4). t(2,1). t(3,1). t(4,1). t(5,1). t(6,1). t(7,1). t(8,1).
l(1). l(2). l(3). l(4). l(5).
```

Predicates `lj`/3, `not`/1, and `min`/3 correspond to the DES built-ins. The Prolog program above mimics them by importing a module `des` where they are defined. The definitions resemble the Datalog semantics, and are as follows:

```
lj(A,B,C) :- A, ( B, C ; set_vars_to_null(B) ).

not(G) :- G -> fail ; true.

min(G,X,M) :- retractall_fact(the_min(_)), G, recorda_minimum(X), fail.
min(_,_,M) :- retract_fact(the_min(M)).

recorda_minimum(X) :- retract_fact(the_min(M0)), !,
                      minimum(M0,X,M), asserta_fact(the_min(M)).
recorda_minimum(X) :- asserta_fact(the_min(X)).
```

The goal `set_vars_to_null(G)` unifies all free variables of term `G` with a reserved constant representing the null value. Let this constant be `null`. The goal `minimum(M0,M1,M)` is true if and only if `M` is the minimum of `M0` and `M1`. Predicates `retractall_fact`/1, `retract_fact`/1, and `asserta_fact`/1 are the usual built-ins for dynamic code manipulation.

A first observation is that the coding of `min`/3 is too complex. We simplify it by `min(_,X,X).`, meaning that the type of the minimum is the same as the type of the corresponding argument of the relation. This is enough for our purposes, and sufficient for program analysis to obtain useful information.

The second step is to unfold the above definition of the Datalog built-ins into the given program. Thus, we obtain the code below, which replaces the original definitions of the corresponding predicates.

```
lj_t_on_s(X,Y,V) :- t(X,Y), (s(U,V) , X<U ; U=null, V=null).
not_l(X) :- l(X) -> fail ; true.
min_t_on_first(X,X):- t(X,Y).
```

The program resulting from the partial evaluation is then analysed for types. In our example, part of the output of analysis is as follows:

```
lj_t_on_s(X,Y,V):- true(t10(X)), t(X,Y),
    (s(U,V) , X<U , true(t20(U)) ; U=null, V=null).
t10(1). t20(2). t20(3). t20(4). t20(5). t20(6).
```

Note the `true` predicates in the analysis output. They are used to assert that the information inferred (in our case, types) holds at the given program points. The types are described by Prolog predicates `t10`

and t20. In the example we only show those that are relevant for the optimization of the computation of predicate lj_t_on_s. That is, the call type t10 for predicate t and the success type t20 for the join.

The information from analysis is passed on to the DES compiler, which performs an optimized computation of the join. The overall result obtained is that, for the evaluation of the join lj_t_on_s, only the following sets of tuples are computed by DES upon execution, instead of the original sets of tuples:

```
t(1,1). t(1,4). s(2,2). s(3,2). s(4,2). s(5,2). s(6,2).
```

The way we pass the information to the DES compiler is by means of a final program transformation. Basically, this transformation partially evaluates the relations involved in the compound goals subject to optimization with respect to the types inferred. The original relations are substituted by new relations which satisfy the inferred types. The new relations are then used in the compound goals. The transformation is performed on the original program to maintain the use of the original DES compound goals. Thus, the program at the beginning of this section turns into the following one:

```
main(lj)  :- lj_t_on_s(1,_,_).
main(not) :- not_l(1), not_l(10).
main(min) :- min_t_on_first(1,_).

lj_t_on_s(X,Y,V) :- lj( t_lj(X,Y), s_lj(U,V), X<U ).

not_l(X) :- not( l_not(X) ).

min_t_on_first(X,M) :- min( t_min(X,Y), X, M ).

s_lj(2,2). s_lj(3,2). s_lj(4,2). s_lj(5,2). s_lj(6,2).
t_lj(1,1). t_lj(1,4). t_min(1,1). t_min(1,4). l_not(1).
```

Note that further program specialization is possible. For example, this is the case of the predicate not_l: It could be specialized with respect to the two calls to it in the program. The specialized versions could then be further evaluated: not_l(1) to true and not_l(10) to false. However, in our work, we are not considering specialization (in this fashion) for the time being.

A form of specialization is embedded in the program transformation itself: the relations occurring in compound goals are particularized for each of the compound goals where they occur. This is the case, for example, for the predicate t/2. Alternatively, one could think of skipping this specialization so as to output a single version for each relation involved. In the example, this will amount to delete the predicate t_min/2 and use instead t_lj/2 for both goals to lj and min.

In fact, the implementation developed for testing these techniques uses intermediate predicates as stubs for matching the original calls to t/2, l/1 and s/2 to the types inferred that represent the specialized versions of those predicates. For example, the goal lj(t(X,Y), s(U,V), X<U) is translated into lj('$t_1_s'(X,Y),'$s_2_s'(U,V),X<U), and the specialized versions of t/2 and s/2 are defined as:

```
'$t_1_s'(A,B) :- '$t_1'(A,B), t(A,B).     '$s_2_s'(A,B) :- '$s_2'(A,B), s(A,B).
'$t_1'(A,B) :- rt5(B), rt0(A).            '$s_2'(A,B) :- rt22(B), rt27(A).
```

Predicates prefixed with rt correspond to descriptive types inferred by the analysis. The information generated by the analysis is related to variables instead of goals, and therefore '$t_1'/2 and '$s_2'/2

are in fact over-approximations of the specialized versions of t/2 and s/2. In this implementation, this loss of precision has been addressed by means of calls to the original predicates t/2 and s/2 in another layer of stub predicates ('$t_1_s'/2 and '$s_2_s'/2). Although a simple solution for this problem, it implies some efficiency loss.

5 Experiments

For testing the techniques described in previous sections, three sample programs have been developed. We have used the type inference analysis of logic programs described in Section 4.2 for obtaining information regarding Datalog programs. Therefore, we restrict the experiments to those that allow optimization using such kind of information.

Benchmarks have been run five times, taking their execution times, removing the highest and lowest results, and computing the average on the three remaining execution times. Table 1 shows the speed-up obtained with the optimizations presented in this paper: the ratio between the execution time of the original program and the execution time of the optimized program.

% filtered tuples	lj/3	not/1	count/3
92%	3.33	2.46	1.33
90%	2.24	1.93	1.18
88%	1.60	1.80	0.90
86%	1.17	1.61	-
84%	0.87	1.51	-
63%	-	1.02	-
62%	-	0.98	-

Table 1: Speed-up with Optimizations (figures show the ratio between the execution times of the original and the optimized programs.)

The first experiment checks how outer joins perform when optimizing a program. The program evaluated is as follows:

```
filter(1). filter(5). filter(15). filter(35). filter(45). filter(55).
main :- test1(X,V).
test1(X,V) :- filter(X), lj(t(X,Y),s(Y,V),V < 10).
s(1,2). s(2,3). s(3,4). ... s(49,50). s(50,51).
t(1,0). t(2,1). t(3,2). ... t(49,48). t(50,49).
```

Predicate filter/1 acts as a filter on the tuples generated by the outer join lj(t(X,Y), s(Y,V), V < 10). Predicates s/2 and t/2 contain fifty tuples as shown.

This program has been tested with a different number of tuples for predicate filter/1 in order to check how the optimization behaves with respect to the number of tuples filtered. The Datalog solving procedure first computes the complete answer sets for both filter(X) and lj(...), and then composes both results to produce the result of predicate test1/2. With the optimization for outer joins, t/2 and s/2 are replaced with specialized versions, according to the information available at compile time for the variables involved in the goal lj(...).

Experimental results show that the performance of this optimization depends very much on the number of tuples filtered by filter/1. If this predicate filters more than 85% of the tuples (i.e., 15% of

the tuples of `lj(...)` satisfy also `filter(X)`), then the optimization improves the original result. The improvements obtained filtering 90% of the tuples are actually remarkable.

Our second experiment tests how negation can be improved with static analysis information. The sample program is as follows:

```
filter(1). filter(5). filter(15). filter(35). filter(45). filter(55).
main :- test2(X).
test2(X) :- filter(X), not(s(X,X)).
s(1,2). s(2,3). s(3,4). ... s(49,50). s(50,51).
```

We can see in the third column of Table 1 the behaviour of the system depending on the number of tuples of predicate `filter/1`. Since negation computes the complete set of answers of the negated goal, in this example the performance of `not(s(X,X))` directly depends on the number of tuples computed for `s(X,X)`. In this case, the static analysis phase actually generates a set of tuples of the same size of `filter/1`. Therefore, the speed-up for this program is more incremental than the other two cases. In fact, the relative slowdown with respect to the number of clauses involved in `filter/1` is caused by some intermediate predicates introduced in the automatic generation of the optimized program, as explained in Section 4.3.

The third experiment tests the use of aggregate predicates, represented by a program counting the size of a subset of tuples in a given predicate. In this case, the example program is as follows (taken from [32]):

```
filter(sales).
main :- active_employees(D,T).
active_employees(D,T) :- filter(D), count(employee(N,D,S),S,T).
% Relation employee(Name,Department,Salary)
employee(anderson,accounting,1200). ...
```

As in previous tests, the call to the aggregate `count(employee(N,D,S),S,T)` is being filtered by the goal `filter(D)`, which in this case has only one tuple corresponding to `sales` department. The number of tuples that are taken into account at execution time is then filtered to those belonging to that department.

Last column of Table 1 shows the improvement obtained. Only for filterings of 90% or more of the tuples the optimization for this aggregate predicate is more efficient than the original program. In this case the speed-up is not as relevant as in previous tests, mainly because the current implementation described in Section 4.3 introduces some overhead caused by several intermediate predicates. In addition, the computation made by `count/3` predicate is more efficient than what is done in previous examples with `lj/3` and `not/1`.

6 Conclusions and Related Work

We have shown the application of static analysis techniques of logic programs to the optimization of Datalog queries via program transformations. This technique has been focused on the optimization of queries involving certain built-ins which are hard to compute because of the underlying deductive engine of a particular implementation of Datalog. Information gathered from an (abstract-interpretation-based) descriptive type static analysis reveals to be crucial to prune computations, notably when strata are involved. In fact, filtering tuples that actually will be relevant for a computation play an important role

with respect to run-time performance, as it has been confirmed by some preliminary and promising results we have presented. Note that the figures about the improvement in efficiency become of paramount importance when large databases are considered, since in-memory requirements hugely decrease.

Although not shown in the examples, the type inference is weak when aliasing is involved, so that more accurate analyzers can be considered. In addition, the technique here presented to take advantage of inferred information can also be used for a more general transformation scheme which can consider partial evaluation for arbitrary predicates. To this end, computations could be faster since smaller answer sets can be handled, by prefiltering relations with the type information.

A number of improvements can be made to the preliminary implementation described in this paper. The compile-time transformation phase can reduce the number of intermediate predicates required for generating specialized versions of predicates. This enhancement should improve the efficiency of this approach, since it is an important source of execution slow-down. This is confirmed by the experimental results since they highlight that a transformed program is more efficient only when there is a relevant reduction in the number of tuples considered for compound queries. A threshold can be used at compile-time in order to specialize only those goals that are guaranteed to gain performance.

Another relevant issue that needs to be solved is related to scalability. The analysis task can be more costly than actually performing the original queries if either the sets of data are large, or the analyzer can simply run out of resources, or the number of specialized versions of predicates grows out of control. In these cases, as well as those optimizations that do not guarantee to speed up the queries, a different approach must be taken. This technique can be selectively applied to some predicates only, in order to control the size of the transformed program, due to a possibly large number of specialized versions of predicates. Another interesting alternative is to use techniques based on sampling, as the ones presented in [1]. This relies on maintaining small representative samples of predicates to keep the analysis effective but, unfortunately, no details are provided.

Techniques such as [31] reuses aggregate selection to Datalog, which roughly consists of pushing selections in the execution plan to reduce the amount of tuples involved in aggregates. Other current systems as the one reported in [28] includes aggregates under XY-stratification, a local and compile-time form of stratification, as well as monotonic aggregates, but reporting no optimizations. With respect to type analysis, other techniques can be considered instead of the Ciao's preprocessor at hand. For instance, [33] includes efficient (PTIME) and sound (but not complete) type-checking algorithms specifically for Datalog. The work [18] introduces a type inference for Datalog and its application to query optimization as type erasure, which saves calls, but metapredicates are not discussed.

Acknowledgements

Work partially supported by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services, Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R) and Madrid regional project N-GREENS Software-CM (S2013/ICE-2731). Also thanks to referees for their suggestions for improvements.

References

[1] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen & Geoffrey Washburn (2015): *Design and Implementation of the LogicBlox System*. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, ACM, New

York, NY, USA, pp. 1371–1382, doi:10.1145/2723372.2742796. Available at http://doi.acm.org/10.1145/2723372.2742796.

[2] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang & Carlo Zaniolo (2003): *The Deductive Database System LDL++*. *TPLP* 3(1), pp. 61–94.

[3] Moritz Becker, Cedric Fournet & Andrew Gordon (2007): *Design and Semantics of a Decentralized Authorization Language*. In: *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, IEEE Computer Society, Washington, DC, USA, pp. 3–15.

[4] Catriel Beeri & Raghu Ramakrishnan (1987): *On the Power of Magic*. In: *Journal of Logic Programming*, pp. 269–283.

[5] Andrea Calì, Georg Gottlob & Thomas Lukasiewicz (2009): *Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints*. In: *ICDT*, pp. 14–30.

[6] Chiara Cumbo, Wolfgang Faber, Gianluigi Greco & Nicola Leone (2004): *Enhancing the Magic-Set Method for Disjunctive Datalog Programs*. In: *In Proc. 20th International Conference on Logic Programming (ICLP 04), LNCS 3132*, Springer, pp. 371–385.

[7] P.W. Dart & J. Zobel (1992): *A Regular Type Language for Logic Programs*. In: *Types in Logic Programming*, MIT Press, pp. 157–187.

[8] C J Date (2009): *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA.

[9] Hendrik Decker (1989): *The Range Form of Databases and Queries or: How to Avoid Floundering*. In Johannes Retti & Karl Leidlmair, editors: *5. ÃÖsterreichische Artificial-Intelligence-Tagung, Informatik-Fachberichte* 208, Springer Berlin Heidelberg, pp. 114–123.

[10] Suzanne W. Dietrich (1987): *Extension Tables: Memo Relations in Logic Programming*. In: *SLP*, pp. 264–272.

[11] Richard Fikes, Patrick J. Hayes & Ian Horrocks (2004): *OWL-QL - a language for deductive query answering on the Semantic Web*. *J. Web Sem.* 2(1), pp. 19–29.

[12] Sergio Greco, Irina Trubitsyna & Ester Zumpano (2005): *NP Datalog: A Logic Language for NP Search and Optimization Queries*. *Database Engineering and Applications Symposium, International* 0, pp. 344–353.

[13] Manuel V. Hermenegildo, F. Bueno, M. Carro, Pedro López, José. F. Morales & Germán. Puebla (2008): *An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy*. In Pierpaolo Degano, Rocco De Nicola & Jose Meseguer, editors: *Festschrift for Ugo Montanari*, *LNCS* 5065, Springer-Verlag, pp. 209–237.

[14] Manuel V. Hermenegildo, Germán. Puebla, Francisco Bueno & Pedro López-García (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. *Science of Computer Programming* 58(1–2), pp. 115–140.

[15] Matthias Jarke, Manfred A. Jeusfeld & Christoph Quix (2008): *ConceptBase V7.1 User Manual*. Technical Report, RWTH Aachen.

[16] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin & Christopher Unkel (2005): *Context-sensitive program analysis as database queries*. In: *PODS*, pp. 1–12.

[17] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri & Francesco Scarcello (2006): *The DLV system for knowledge representation and reasoning*. *ACM Trans. Comput. Log.* 7(3), pp. 499–562.

[18] Oege de Moor, Damien Sereni, Pavel Avgustinov & Mathieu Verbaere (2008): *Type Inference for Datalog and Its Application to Query Optimisation*. In: *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, ACM, New York, NY, USA, pp. 291–300, doi:10.1145/1376916.1376957. Available at http://doi.acm.org/10.1145/1376916.1376957.

[19] Susana Nieva, Fernando Sáenz-Pérez & Jaime Sánchez (2015): *HR-SQL: An SQL Database System with Extended Recursion and Hypothetical Reasoning.* Ongoing Work. In: *XV Jornadas sobre Programación y Lenguajes, PROLE2015 (SISTEDES)*, pp. 1–15.

[20] G. Ramalingam & Eelco Visser, editors (2007): *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. ACM.

[21] Prasad Rao, Konstantinos Sagonas, Terrance Swift, David Warren & Juliana Freire (1997): *XSB: A System for Efficiently Computing Well-Founded Semantics*. In: *In Proceedings of the 4th International Conference on Logic Programming and NonMonotonic Reasoning (LPNMR'97*, Springer, pp. 430–440.

[22] Peter Revesz (2002): *Introduction to Constraint Databases*. Springer-Verlag New York, Inc., New York, NY, USA.

[23] Royi Ronen & Oded Shmueli (2009): *Evaluating very large datalog queries on social networks*. In: *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, ACM, New York, NY, USA, pp. 577–587.

[24] Fernando Sáenz-Pérez (2011): *DES: A Deductive Database System*. *ENTCS* 271, pp. 63–78.

[25] Fernando Sáenz-Pérez (2012): *Outer Joins in a Deductive Database System*. *ENTCS* 282(0), pp. 73 – 88.

[26] Fernando Sáenz-Pérez (2012): *Tabling with Support for Relational Features in a Deductive Database*. *ECEASST* 55.

[27] Konstantinos Sagonas, Terrance Swift & David S. Warren (1994): *XSB as an efficient deductive database engine*. In: *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, ACM, pp. 442–453, doi:http://doi.acm.org/10.1145/191839.191927.

[28] Alexander Shkapsky, Kai Zeng & Carlo Zaniolo (2013): *Graph Queries in a Next-generation Datalog System*. *Proc. VLDB Endow.* 6(12), pp. 1258–1261, doi:10.14778/2536274.2536290. Available at `http://dx.doi.org/10.14778/2536274.2536290`.

[29] Hisao Tamaki & Taisuke Sato (1986): *OLD resolution with tabulation*. In: *Proceedings on Third international conference on logic programming*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 84–98.

[30] Jeffrey D. Ullman (1995): *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press.

[31] Mengmeng Liu nad Zachary G. Ives & Boon Thau Loo: *Query Optimization as a Datalog Program*.

[32] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian & Roberto Zicari (1997): *Advanced Database Systems*. Morgan Kaufmann.

[33] David Zook, Emir Pasalic & Beata Sarna-Starosta (2009): *Typed Datalog*. In: *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, PADL '09, Springer-Verlag, Berlin, Heidelberg, pp. 168–182, doi:10.1007/978-3-540-92995-6_12. Available at `http://dx.doi.org/10.1007/978-3-540-92995-6_12`.