Using Database Optimization Techniques for Nonmonotonic Reasoning*

Wolfgang Faber Nicola Leone Cristinel Mateis Gerald Pfeifer †

Institut für Informationssysteme E184/2, TU Wien
A-1040 Wien, Austria

{faber,leone,mateis,pfeifer}@dbai.tuwien.ac.at

Abstract

In this paper a program rewriting technique for disjunctive datalog is proposed, which descends from query optimization techniques in relational algebra and reduces the size of the ground instantiation of a program in many cases. As a consequence, the time for generating the ground instantiation and the time for subsequent operations on the ground program is shortened.

A part of this technique has already been implemented as a preprocessing step in the Disjunctive Deductive Database System `dlv`. Using a recently published application, we show that a significant reduction of the ground program size and a tremendous overall speedup can be achieved.

1 Introduction

`dlv` is a Disjunctive Deductive Database System implementing the Consistent Answer Sets Semantics. Like similar systems in the area of non-monotonic reasoning [6], the kernel modules of `dlv` operate on a ground instantiation of the input program, i.e., a program that does not contain any variables, but is (semantically) equivalent to the original input [1].

Therefore, an efficient instantiation procedure is of utmost importance. The efficiency of an instantiation procedure can be measured in terms of the size of its output (i.e., the number of rules in the instantiated program resp. the size of these rules) and the time needed to generate this instantiation.

The authors have already presented several techniques [1, 2] which improve along these lines. In this paper we propose a further optimization technique that descends from query optimization techniques in the field of relational databases.

To give an intuition, consider the rule r_1:

```
p(X) :- r(X,Y,Z), q(Z,V,S), V < S.
```

Deductive database systems, based on a bottom-up computational model, evaluate the relational algebra expression corresponding to the body of the rule and assign the result of the evaluation to relation corresponding to the head predicate [7].[1] The relational algebra expression corresponding to the above rule is

$$\mathrm{PROJ}_{\$1}\mathrm{SEL}_{\$4<\$5}[\mathrm{R\ JOIN\ Q}],$$

where R and Q are the relations correponding to predicate r and q, respectively, and JOIN denotes natural join (on common variables). This relational algebra expression can be evaluated more efficiently by "pushing down" projections and selections. Indeed, any relational database system would evaluate the following relational algebra expression which is equivalent to the original one.

$$\mathrm{PROJ}_{\$1}\ [\ [\mathrm{PROJ}_{\$1,\$3}\mathrm{R}]\ \mathrm{JOIN}\ [\mathrm{PROJ}_{\$1}\mathrm{SEL}_{\$2<\$3}\mathrm{Q}]\ ].$$

In this paper, we propose a program rewriting technique, which simulates this "push down" of projections and selections of relational algebra. It is worthwhile noting that the usefulness of "pushing down" projections and selections for the evaluation of traditional (nondisjunctive stratified) deductive databases is well known and has already been implemented in several deductive database systems [7]. The novelty of our approach is the use a "push down" technique in the process of program instantiation for nonmonotonic disjunctive databases, in order reduce the size of the instantiated program. For instance suppose that r and q, in the above example, are defined by disjunctive rules. Then, the ground instantiator of `dlv` must generate the ground instances of r_1. To this end, a preprocessing step rewrites r_1 in the following three rules:

```
p(X) :- r'(X,Z), q'(Z).
r'(X,Z) :- r(X,Y,Z).
q'(Z) :- q(Z,V,S), V < S.
```

*This work was supported by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.

†Please address correspondence to this author.

[1] `dlv` proceeds exactly this way, if r and q are either base predicates or normal stratified predicates. Otherwise, a similar process is used to generate the ground instances of the rule.

The instantiation process is then applied on the rewritten program, which has exactly the same answer sets of the original one, provided that the primed predicates are ignored (in `dlv` they are filtered out in the output).

The main advantages we obtain are (i) the speed-up of the evaluation of normal stratified programs (these program are completely solved/evaluated by dlv's instantiation procedure; the ground instantiation is not generated at all in this case); (ii) the speed-up of the instantiation process for general (disjunctive or unstratified) programs, and, most importantly, (iii) the drastic reduction of the size of the ground instantiation (for general programs), which dramatically improves overall system's performance, as evidentiated in Section 5.

This paper is structured as follows: The next section will provide a quick overview of the language of `dlv`, followed by a detailed description of our proposed optimization in Section 3. We will then present an interesting application for `dlv` from the domain of planning and conclude with a couple of benchmarks and a short analysis.

2 The Language of `dlv`

An *atom* is an expression of the form $a(t_1, \ldots, t_f)$ where a is the predicate name of the atom and each t_i $(1 \leq i \leq f)$ is either a constant or a variable. A literal is an expression of the form L or `not` L, where `not` is the *negation as failure* symbol, and L is an *extended atom*, i.e., either an atom A, or an atom preceded by the *strong negation* symbol '$-$' ($-A$).

A `dlv` *program* is a set of rules and constraints. A *rule* r has the form

$$a_1 \vee \ldots \vee a_m \texttt{ :- } b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_n.$$

where $m \geq 1$, $n \geq 0$ and the a_i and b_j are extended atoms.

The part to the left of `:-` is called the *head* of the rule, the part to the right is the *body*.

A *constraint* is a rule with an empty head ($m = 0$). A rule with $m = 1$ and an empty body is called a *fact*, and we usually omit "`:-`" in this case.

In addition to user defined atoms as described above, `dlv` also supports various *built-in predicates* like `<`, `=`, and `!=` (inequality).

Constants are strings starting with a lower-case letter, while variables are strings starting with an upper-case letter. Anonymous variables are denoted by "`_`". Each occurrence of an anonymous variable stands for a variable that is different from all others in the entire rule.

Ground queries of the form

$$b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_n?$$

are allowed and are translated to the following constraints internally:

$$\texttt{:- not } b_1.$$
$$\vdots$$
$$\texttt{:- not } b_k.$$
$$\texttt{:- } b_{k+1}.$$
$$\vdots$$
$$\texttt{:- } b_n.$$

For example, the query `a, not b?` will generate the two constraints `:- not a.` and `:- b.`

The semantics of a `dlv` program is given by those *consistent Answer Sets* [5] satisfying all constraints. Each program can have no, one or several consistent Answer Sets.

A nice example for the power of this formalism is Graph 3-Colorability: Given an undirected graph, represented by facts of the form `edge(_,_)` and `node(_)`, assign each node one of three colors such that no two adjacent nodes have the same color. The simple program in Figure 1 computes the legal 3-colorings of the graph.

```
% Each node is either red, green, or blue.
col(X,r) v col(X,g) v col(X,b) :- node(X).

% Two nodes on the same edge
% must not have the same color.
:- edge(X,Y), col(X,C), col(Y,C).
```

Figure 1: Graph 3-Coloring

3 Relational Optimization

As we have outlined in Section 1, it is important to have a ground instantiation (grounding), which is as small as possible and generated in as little time as possible. In this section we will present the details of our proposed optimization.

Basically, rules of the non-ground program that meet certain syntactic conditions are transformed and additional rules are added to the non-ground program. The answer sets of the original (non-ground) program are exactly the answer sets of the transformed program, after the instances of the auxiliary predicates defined by the added rules have been eliminated.

Consider a non-ground rule r, which contains an atom $p(X_1, \ldots, X_n)$ in its body, and a variable X_i, $1 \leq i \leq n$, which does not appear anywhere in r except in p. According to the technique described in [1, 2], the grounding of r proceeds by matching the atoms in the positive body of r one by one with their instances and binding the free variables accordingly in each step, as in a nested loop join algorithm for databases. If the matching of the current predicate fails, we backtrack to the previous predicate trying to match it with another instance.

Clearly, X_i does not influence the matching of the atoms different from p nor the instances obtained for the atoms in the head of r. Thus, we can eliminate X_i by projecting p on all variables X_k, $k \neq i$, obtaining an auxiliary predicate p'. That is, we add a new (non-ground) rule

$$p'(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) \text{ :- } p(X_1, \ldots, X_n).$$

We then substitute p by p' in the body of r. In this way the generation of ground instances of r which differ only on the binding of X_i is avoided.

For simplicity, we described here the case where only one variable is eliminated from p, but the optimization can be easily generalized (and is implemented in `dlv`) to the case where several variables from p do not appear anywhere else in r.

A further generalization of this optimization technique is to project groups of atoms instead of only single atoms. Consider for instance the rule r_2,

```
a(X) v b(Y) :- c(X,Z,W), d(Z,Y), e(Y,W).
```

and note that the variable `Z` appears in both `c` and `d` but not in `e`, `a` or `b`.

We add the rule

```
f(X,Y,W) :- c(X,Z,W), d(Z,Y).
```

and substitute r_2 by a new rule r'_2,

```
a(X) v b(Y) :- f(X,Y,W), e(Y,W).
```

The ground instances are generated faster for r'_2 because generation of useless ground instances is avoided by eliminating `Z` from the rule body.

The optimization can be further improved by exploiting built-in literals. Built-in literals usually impose a relation (e.g., equality) between their arguments. Therefore, these literals narrow the set of ground instances to be matched with the other body literals. Consider for instance the rule r_3,

```
p(X,Y,Z) :- ..., q(X,Y,Z), ..., X < Y, ....
```

The set of ground instances of `q` can be narrowed before starting to ground r_3 by adding a new rule r'_3,

```
q'(X,Y,Z) :- q(X,Y,Z), X < Y.
```

and replacing r_3 by

```
p(X,Y,Z) :- ..., q'(X,Y,Z), ....
```

In this way, during the grounding of r'_3, we avoid useless relational join operations after matching `q` and before finding out that the arguments of the matched instance do not satisfy the inequality relation imposed by the built-in literal `X < Y`.

Another generalization of this technique considers groups of atoms and can select more than one built-in literal to narrow the ground instances to be matched with body literals. Consider for instance the rule r_4,

```
a(X) v b(Y) :-
   c(X,Z), d(V,W), e(Y,V,W), Z < W.
```

We add the rule

```
f(X,V,W) :- c(X,Z), d(V,W), Z < W.
```

and we substitute r_4 by the new rule r'_4,

```
a(X) v b(Y) :- f(X,V,W), e(Y,V,W).
```

The disadvantage of grounding r_4 instead of r'_4 is that, after having matched `c` and `d`, we anyway match `e`, even if the variables `Z` and `W` bound by `c` and `d` do not match with the built-in literal `Z < W` and no ground instance can be obtained at all.

As already outlined in Section 1, the main advantages of the proposed optimization technique are (i) the speed-up of the evaluation of normal stratified programs, (ii) the speed-up of the instantiation process for general (disjunctive or unstratified) programs, and (iii) the drastic reduction of the size of the ground instantiation (in general). It is worth noting that the technique applies to both rules and constraints (the latter are just rules with an empty head).

Let $\mathcal{P}$ be a (non-ground) program and $\mathcal{P}'$ the program obtained from $\mathcal{P}$ by applying the rewriting optimization technique. Given an answer set M for $\mathcal{P}'$, $\pi(M)$ is the set of literals obtained from M by eliminating all the auxiliary literals, i.e., $\pi(M)$ is the set of literals without all atoms which were derived from the rules introduced by the optimization technique. $\mathcal{P}'$ can be used in place of $\mathcal{P}$ in order to evaluate answer sets of $\mathcal{P}$. The result supporting the above statement is the following:

Theorem 1 *For each answer set M' for $\mathcal{P}'$, $\pi(M')$ is an answer set for $\mathcal{P}$. Moreover, for each answer set M for $\mathcal{P}$ there exists an answer set M' for $\mathcal{P}'$ such that $\pi(M') = M$.*

4 An Application – Planning

An interesting application for DDLPs is planning, from AI. Most of the literature on planning does not consider representations employing disjunction, but recently research has started to consider this novel approach [3].

As an example domain we have chosen the "classic" blocksworld. In this section we will provide a natural encoding of this problem in the language of `dlv`, which we will then use for benchmarking in Section 5.

A "blocksworld" consists of one table and an arbitrary number of labeled cubic blocks. Both the table and the blocks are referred to as `locations`:

```
location(table).
location(L) :- block(L).
```

In addition to the fixed program (query) shown below, information on the actual blocks is provided either in the form of facts in our language or stored in a database.

The state of the blocksworld at a particular time can be fully specified by the relation `on(B,L,T)`, which specifies that block `B` resides on location `L` at time `T`.

The concept of time in the current context is a discrete one, that is, we deal with a limited number of fixed time steps which are represented by means of integer built-in predicates and constants. The first time is always `0`, and the last time `k` is specified when invoking `dlv` via the command-line option `-N=k`.

The predicate `#int(T)` represents all numbers in [0..`k`], `#succ(T,T1)` defines the successor relation in the same interval, and the constant `#maxint` is substituted by `k`:

```
time(T) :- #int(T).
next(T,T1) :- #succ(T,T1).
lasttime(#maxint).
```

In the blocksworld domain, we have only one kind of action, namely moving a block from one location to another one. A move is started at one point in time, and it is completed before the next time. At any time `T`, the action of moving a block `B` to location `L` may be initiated (`move(B,L,T)`) or not (`-move(B,L,T)`):

```
move(B,L,T) v -move(B,L,T) :-
   block(B), location(L), time(T).
```

A block may be moved only if it is not blocked by another block residing on it:

```
:- move(B,L,T), on(B1,B,T).
```

Similarly, if the block is moved onto another block, no block may reside on the latter. Note that an arbitrary number of blocks may be moved onto the table:

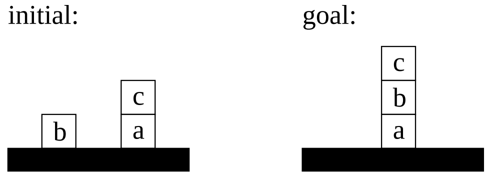
```
:- block(B1), move(B,B1,T), on(B2,B1,T).
```

The effect of a `move` action is that the moved block is at the new location at the next point in time:

```
on(B,L,T1) :- move(B,L,T), next(T,T1).
```

We do not allow for simultaneous moves, that is, at most one action may be initiated at any time. In addition, no action may be initiated at the last timeslot:

```
:- move(B,L,T), move(B1,L1,T), B != B1.
:- move(B,L,T), move(B1,L1,T), L != L1.
:- move(B,L,T), lasttime(T).
```




```
% Define the involved blocks.
block(a).  block(b).  block(c).

% Specify the initial and goal situations.
on(a,table,0).  on(b,table,0).  on(c,a,0).
on(a,table,#maxint), on(b,a,#maxint),
on(c,b,#maxint)?
```

Figure 2: The Sussman anomaly.

The `on` relation is inertial. This means that, unless it is changed by an action, it remains stable over the time.

```
on(B,L,T1) :-
   on(B,L,T), next(T,T1), not -on(B,L,T1).
-on(B,L,T1) :-
   -on(B,L,T), next(T,T1), not on(B,L,T1).
```

Since the actual moves take place between the time slots, any block may only be at one location at any given time.

```
-on(B,L1,T) :-
   on(B,L,T), location(L1), L != L1.
```

And finally, a block is never allowed to be on itself:

```
:- on(B,B,T).
```

For a problem instance, the involved blocks and the initial state (describing where each block is located) are specified by facts in our language or by a database. The task then is to find a sequence of moves (a *plan*) such that the goal state (which is given by a query) is reached by applying this sequence of actions to the initial state.

One classical instance is the so-called Sussman anomaly, depicted in Figure 2. When we look for a plan of length 3 (by specifying `-N=3` on the command-line) for this instance, exactly one is found, and indeed this is the only valid plan:

```
move(c,table,0), move(b,a,1), move(c,b,2)
```

5 Benchmarks and Conclusion

The "May 28 1999" release of `dlv` already includes a simple version of our new relational optimization which considers only single literals, i.e. none of the more involved techniques described in section 3 is applied yet. The command-line options for enabling and disabling it are `-OR` and `-OR-`, respectively. By default, it is enabled. Based on [3], we chose several blocksworlds instances of increasing size and performed benchmarks on a PentiumII/333 with 128MB of RAM running FreeBSD 3.1.

Figure 3 has detailed data on the sizes of the ground programs encountered without resp. with this optimization, while Figure 4 provides execution times for finding the first answer set.

Problem		ground rules	ground constraints
P1	–OR–	544	3085
	–OR	528	318
P2	–OR–	1445	10598
	–OR	1210	726
P3	–OR–	6272	82013
	–OR	3904	2306
P4	–OR–	16247	311121
	–OR	8074	4735
P5	–OR–	20207	380861
	–OR	9922	5813

Figure 3: Blocksworld: Grounding

Problem	Blocks	Steps	With –OR–	With –OR
P1	4	4	0.5s	0.2s
P2	5	6	2.4s	1.0s
P3	8	8	50.3s	9.5s
P4	11	9	773.1s	56.4s
P5	11	11	—	141.2s

Figure 4: Blocksworld: Execution Times

It is evident that already the simple form of our proposed optimization provides a significant reduction in the size of the ground program as well as a tremendous performance boost. We are currently working on the implementation of the more involved general version of this optimization, which should provide further improvements for many programs.

Up-to-date information on `dlv` as well as binaries for common platforms, a full-fledged manual and some test instances are available at the project web site [4].

Acknowledgements

We thank the reviewers for their valuable comments.

References

[1] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Nonmonotonic Reasoning. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Berlin, 1997. Springer.

[2] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. Progress Report on the Disjunctive Deductive Database System `dlv`. In T. Andreasen, H. Christiansen, and H. L. Larsen, editors, *Proceedings International Conference on Flexible Query Answering Systems (FQAS '98)*, pages 145–160, Roskilde University, Denmark, May 1998.

[3] E. Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft, 1999.

[4] W. Faber and G. Pfeifer. `dlv` homepage. `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[5] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[6] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997.

[7] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.