DBAI

TECHNICAL REPORT

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

Efficient Instantiation of Disjunctive Databases

DBAI-TR-2001-44

Wolfgang Faber **Nicola Leone** **Simona Perri**
Gerald Pfeifer

DBAI TECHNICAL REPORT
NOVEMBER, 2001

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18492
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

TU
TECHNISCHE UNIVERSITÄT WIEN

DBAI TECHNICAL REPORT

DBAI TECHNICAL REPORT DBAI-TR-2001-44, NOVEMBER, 2001

Efficient Instantiation of Disjunctive Databases

Nicola Leone, Simona Perri [1] **Wolfgang Faber, Gerald Pfeifer**[2]

Abstract. Most deductive database systems are endowed with an instantiation module. The instantiator generates a new program which is equivalent to the input program, but does not contain any variables (i.e., it is ground). The instantiation process may be computationally expensive in some cases, and the instantiator is crucial for the efficiency of the entire ASP system. In this report we describe the instantiation procedure of DLV system, which is one of its strong points. Using differential and other advanced database techniques together with suitable data structures, the DLV instantiator efficiently generates a ground instantiation of the input that has the same stable models as the full program instantiation, but is much smaller in general. Moreover, in case of normal stratified programs, it already computes the single stable model without producing any instantiation.

[1]Department of Mathematics, Università della Calabria, Via Pietro Bucci, 30B, I-87036 Rende (CS), Italy. Email: {leone, perri }@mat.unical.it

[2]Institut für Informationssysteme, TU Wien A-1040 Wien, Austria. Email: {faber, pfeifer }@ dbai.tuwien.ac.it

1 Introduction

Deductive databases extend relational database systems by the inferential power of logic programming. Their study has been one of the major activities of database researchers in the second half of the eighties. Besides important theoretical results, this study has led to the implementation of a number of deductive database systems supporting logic programming and its extensions as the query language [CCCR$^+$90, CGK$^+$90, LR95, PDR91, RSS92].

Recently, the idea of incorporating disjunction in the deductive database languages has stimulated a renewed interest in this area, since deductive databases allowing disjunction, called Disjunctive Deductive Databases (DDDBs), seem to be well-suited to perform nonmonotonic reasoning tasks, which are strongly needed in the field of Artificial Intelligence. Thus, during the last years, much research has been done concerning semantics and complexity of Disjunctive Deductive Databases. Interesting studies on the expressive power of DDDBs [EGM97] have also shown that some concrete real world problems cannot be represented by (v-free) deductive databases, while they can be represented by DDDBs.

This technical report focuses on the Disjunctive Database System DLV. Like similar systems in the area of non-monotonic reasoning, the kernel modules of DLV operate on a ground instantiation of the input program, i.e., a program that does not contain any variable, but is (semantically) equivalent to the original input [ELM$^+$97]. Indeed, any given program P first undergoes the so-called instantiation process, that computes from P such instantiation. Since this instantiation phase may be computationally very expensive, having a good instantiation procedure (also called instantiator) is a key feature of DDDBs systems. 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 and the size of these rules, respectively) and the time needed to generate this instantiation.

The size of the generated instantiation is important because it strongly influences the computation time spent by the other modules of the system. A slower instantiation procedure generating a smaller grounding may be preferable to a faster one generating a large grounding.

The main reason of large groundings even for small input programs is that each atom of a rule in $\mathcal{P}$ may be instantiated to many atoms in Herbrand Base of $\mathcal{P}$, which leads to combinatorial explosion. However, most of these atoms may not be derivable whatsoever, and hence such instantiations do not render applicable rules. A good instantiator should generate ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$.

In this technical report we describe the instantiation procedure of DLV, which is considered as a strong point of system. In order to evaluate efficiently stratified programs (components), DLV uses an improved version of the generalized semi-naive technique [Ull89] implemented for the evaluation of linear and non-linear recursive rules. If the input program is normal (i.e., $\vee$-free) and stratified, the instantiator evaluates completely the program and no further module is employed after the grounding; the program has a single stable model, namely the set of the facts and the atoms derived by the instantiation procedure. If the input program is disjunctive or unstratified, the instantiation procedure cannot evaluate completely the program. However, the optimization techniques mentioned above are useful to generate efficiently the instantiation of the non-monotonic part of the program.

Moreover, we present an optimization technique, descending from query optimization techniques in relational algebra, which allows to further reduce the size of the produced instantiation.

The remainder of the report is structured as follows: in Section 2 we give some basic notions of Disjunctive Databases; in Section 3, we give an overview of the architecture of the DLV instantiator; in Section 4 we describe The Dependency Graph Handler, a module of the DLV instantiator which identifies dependencies between predicates needed for the instantiation process; in Section 5 we describe the core procedure of the instantiator; in Section 6 we present a rewriting technique which often allows for a smaller and faster instantiation; finally, Section 7 is devoted to experimental results, and Section 8 draws our conclusions.

2 Disjunctive Databases

In this section, we provide a formal definition of the syntax and semantics of disjunctive databases.

2.1 Syntax

A variable or constant is a *term*. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms; for nullary predicates ($n = 0$), we usually omit the parentheses. A *literal* is either a *positive literal* p or a *negative literal* $\neg p$, where p is an atom. Given a *literal* l, we define $\neg.l$ as $\neg l$ if l is a positive literal or as p if l is a negative literal $\neg l$. Similarly, given a set of literals L, $\neg.L$ denotes $\{\neg.l | l \in L\}$.

A *(disjunctive) rule* r is a syntactic of the following form:

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, \neg b_{k+1}, \cdots, \neg b_m. \qquad n \geq 1,\ m \geq k \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \neg b_{k+1}, ..., \neg b_m$ is the *body* of r. If the body is empty, we usually also omit the "$\leftarrow$" when writing a rule.

$H(r)$ denotes the set $\{a_1, ..., a_n\}$ of the head atoms, and $B(r)$ the set $\{b_1, ..., b_k, \neg b_{k+1}, ..., \neg b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$.

A *disjunctive datalog program (or disjunctive database, DDB)* $\mathcal{P}$ is a finite set of rules. A $\neg$-free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it. A ground program is also called a *propositional* program.

A predicate occurring only in *facts* (rules of the form $a \leftarrow$) is referred to as *EDB* predicate, all others as *IDB* predicates. The set of facts in which *EDB* predicates occur in, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

Please note that we make frequent use of rules without a head $\leftarrow l_1, \ldots, l_n.$, called *constraints*, which are a shorthand for $false \leftarrow l_1, \ldots, l_n.$, and it is also assumed that a rule $bad \leftarrow false, \neg bad$ is in the DDB, where $false$ and bad are special symbols appearing nowhere else in the DDB. So, intuitively, the body of a constraints must not be true in any stable model.

2.2 Semantics

Let $\mathcal{P}$ be a program. The *Herbrand universe* $U_{\mathcal{P}}$ of a DDB $\mathcal{P}$ is the set of constants that appear in the program.[1]

The *Herbrand base* $B_{\mathcal{P}}$ of a DDB $\mathcal{P}$ is the set of all possible ground atoms that can be constructed from the predicates appearing in the rules of $\mathcal{P}$ and the terms occurring in $U_{\mathcal{P}}$. Note that for disjunctive databases, $U_{\mathcal{P}}$ and $B_{\mathcal{P}}$ are always finite.

Given a rule r occurring in a DDB, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a mapping from the variables occurring in r to the terms in $U_{\mathcal{P}}$. We denote by $ground(\mathcal{P})$ the set of all the ground instances of the rules occurring in $\mathcal{P}$.

An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_{\mathcal{P}}$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal $\neg A$ is *true* w.r.t. I if A is false w.r.t. I; otherwise $\neg A$ is false w.r.t. I.

Let r be a ground rule in $ground(\mathcal{P})$. The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in ground(\mathcal{P})$ is true w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$.

The first proposal for assigning a semantics to a disjunctive logic program appears in [Min82], which presents a model-theoretic semantics for positive programs. According to [Min82], the semantics of a program $\mathcal{P}$ is described by the set $\mathrm{MM}(\mathcal{P})$ of the minimal models for $\mathcal{P}$. Observe that every program $\mathcal{P}$ admits at least one minimal model, that is, for every program $\mathcal{P}$, $\mathrm{MM}(\mathcal{P}) \neq \emptyset$ holds.

Example 2.1 *For the positive program* $P_1 = \{a \vee b \leftarrow\}$*, the interpretations* $\{a\}$ *and* $\{b\}$ *are its minimal models (i.e.,* $\mathrm{MM}(\mathcal{P}) = \{\ \{a\},\ \{b,\}\ \}$*).*

For the program $P_2 = \{a \vee b \leftarrow;\ b \leftarrow a;\ a \leftarrow b\}$, $\{a, b\}$ *is the only minimal model.* $\Box$

As far as general programs (i.e., programs where negation may appear in the bodies) are concerned, a number of semantics have been proposed [BD95, GL91, Min82, Prz90, Prz91, Prz95, Ros90, Sak89] (see [AB94, Dix95, LMR92] for comprehensive surveys). A generally acknowledged semantics for disjunctive datalog programs is the extension of the stable model semantics [GL88] to take into account disjunction [GL91, Prz91]. Given a program $\mathcal{P}$ and an interpretation I, the *Gelfond-Lifschitz (GL) transformation* of $\mathcal{P}$ w.r.t. I, denoted $\mathcal{P}^I$, is the set of positive rules defined as follows:

$$\begin{aligned}\mathcal{P}^I = \{ \quad & a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k \mid \\ & a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, \neg b_{k+1}, \cdots, \neg b_m \\ & \text{is in } ground(\mathcal{P}) \text{ and } b_i \notin I, \text{ for all } k < i \leq m\}\end{aligned}$$

[1]If no constants appear in the program, then one is added arbitrarily.

Definition 2.1 [Prz91, GL91] Let I be an interpretation for a program $\mathcal{P}$. I is a *stable model* for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model of the positive program $\mathcal{P}^I$). The set of all stable models for $\mathcal{P}$ is denoted by $STM(\mathcal{P})$. □

Clearly, if $\mathcal{P}$ is positive, then $\mathcal{P}^I$ coincides with $ground(\mathcal{P})$. It turns out that for a positive program, minimal and stable models coincide.

3 Architecture of DLV's Intelligent Grounding

The general structure of the Intelligent Grounding (IG) of DLV is depicted in Figure 1.

An input program $\mathcal{P}$ is first analyzed by the parser, which also builds the extensional database from the facts in the program, and encodes the rules in the intensional database in a suitable way. Then, a rewriting procedure (see Section 6), optimizes the rules in order to get an equivalent program $\mathcal{P}'$ that can be instantiated more efficiently and that can lead to a smaller ground program. At this point, another module of the instantiator computes the dependency graph of $\mathcal{P}'$, its connected components, and a topological ordering of these components. Finally, $\mathcal{P}'$ is instantiated one component at a time, starting from the lowest components in the topological ordering, i.e., those components that depend on no other component, according to the dependency graph.

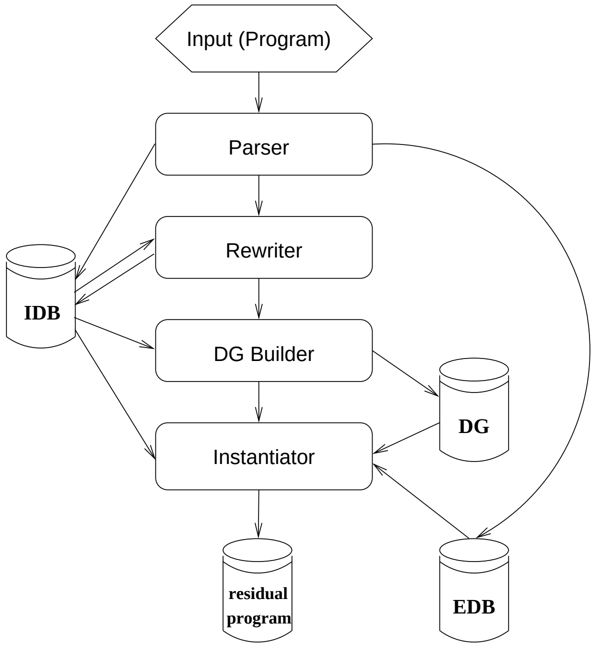

Figure 1: System Architecture

4 Dependency Graph Handler

The Dependency Graph Handler (DGH) builds and handles graphs representing different relations between the predicates of a given program $\mathcal{P}$, plus it singles out and analyzes syntactic modules of $\mathcal{P}$ according to these relations.

The instantiation procedure that we are going to present in the following needs information about (positive) dependencies among the IDB predicates of the input program $\mathcal{P}$. By detecting mutually recursive predicates and by imposing a suitable order on the rules defining them, a ground program Π "equivalent" to $\mathcal{P}$ and $ground(\mathcal{P})$ can be generated which usually avoids a lot of useless work (both during instantiation and, by often being sensibly smaller and simpler, also for later phases).

Given a program $\mathcal{P}$, the most important task performed by the DGH is splitting $\mathcal{P}$ into syntactic modules, which can be evaluated separately [LT94, EGM97]. This can lead to an exponential gain in efficiency, as even in cases where $\mathcal{P}$ itself is not in an efficiently evaluable class of programs, many of these modules (or components) of $\mathcal{P}$ may be. They often fit into known tractable syntactic classes, like stratified normal logic programs, which can be completely evaluated in polynomial time.

We define relations $\prec^+$ and $\prec^-$ between IDB predicates of $\mathcal{P}$ as follows: For any $p, q \in IDB(\mathcal{P})$, $p {\prec^+} q$ (resp., $p {\prec^-} q$) holds if there exists a rule $r \in \mathcal{P}$ such that $p \in H(r)$ and $q \in B^+(r)$ (resp. $q \in \neg.B^-(r)$). In other words, $p {\prec^+} q$ denotes a positive dependency of p on q, while $p {\prec^-} q$ denotes a negative dependency of p on q (where p depends on a negative body literal $\neg q$).

We associate with a program $\mathcal{P}$ two digraphs, $G_\mathcal{P}$ and $G^+_\mathcal{P}$, both of which have the IDB predicates of $\mathcal{P}$ as their nodes.

The set of arcs E of $G_\mathcal{P}$ is the union of two differently labeled sets of arcs E^+ and E^-. For each pair of nodes, q and p, the arc $q \rightarrow p$ is in E^+ (resp. E^-) iff $p {\prec^+} q$ (resp. $p {\prec^-} q$) holds. The labels associated with arcs are useful to single out syntactic features of subprograms like limited use of negation. The set of arcs E of $G^+_\mathcal{P}$ is given by E^+ only. Note that, consequently, $G^+_\mathcal{P}$ is the subgraph of $G_\mathcal{P}$ obtained by removing the arcs of E^- from $G_\mathcal{P}$.

The graph $G_\mathcal{P}$ naturally induces a partitioning of $\mathcal{P}$ into modules (or subprograms) that allows for a modular evaluation. Recall that, we say that a rule $r \in \mathcal{P}$ *defines* a predicate p if $p \in H(r)$. A *module* of $\mathcal{P}$ is the set of rules defining all the predicates contained in a particular maximal strongly connected component (SCC) of $G_\mathcal{P}$. Intuitively, a module includes all rules defining one predicate p together with the rules defining all other predicates that are mutually dependent with p.

We say that a rule $r \in \mathcal{P}$ is *normal stratified* iff (i) r is normal (i.e., disjunction-free), and (ii) there is no cycle in $G_\mathcal{P}$ containing both $H(r)$ and any predicate appearing in $B^-(r)$. A program $\mathcal{P}$ is normal stratified iff all its rules are normal stratified. Note that if $\mathcal{P}$ is stratified, then the maximal strongly connected components of $G_\mathcal{P}$ and $G^+_\mathcal{P}$ coincide.

For any maximal strongly connected component C of $G^+_\mathcal{P}$, we denote by $recursive\_rules_\mathcal{P}(C)$ the set of the rules r from $\mathcal{P}$ such that predicates from C occur both in $H(r)$ and in $B^+(r)$, and by $exit\_rules_\mathcal{P}(C)$ the remaining set of rules r in $\mathcal{P}$ that define a predicate from C. Moreover, we say that a rule r from $\mathcal{P}$ is *total* if either (i) r is a fact, or (ii) r is normal stratified and every body literal is defined only by total rules. A predicate is total if all the rules defining it are total, and intuitively

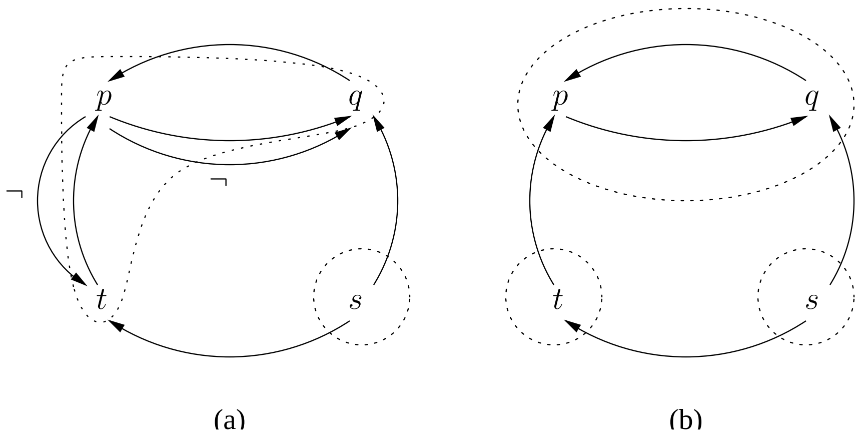

Figure 2: Dependency Graphs (a) $G_{\mathcal{P}}$, and (b) $G^{+}_{\mathcal{P}}$

total predicates are those than can be fully evaluated in a deterministic way (by the instantiator).

The Dependency Graph Handler (DGH) computes the graphs $G_{\mathcal{P}}$ and $G^{+}_{\mathcal{P}}$, and their maximal strongly connected components.

Example 4.1 *Consider the following program $\mathcal{P}$, where a is an EDB predicate:*

$$\begin{array}{rcl@{\qquad\qquad}rcl} p(X,Y) & \leftarrow & q(X), t(Y) & q(X) \vee t(Y) & \leftarrow & s(X), s(Y), \neg p(X,Y) \\ q(X) & \leftarrow & s(X), p(X,Y) & s(X) & \leftarrow & a(X) \end{array}$$

The graphs $G_{\mathcal{P}}$ and $G^{+}_{\mathcal{P}}$ are depicted in Figure 2. The SCCs of $G_{\mathcal{P}}$ are $\{p,q,t\}$ and $\{s\}$. They correspond to the modules $\{p(X,Y) \leftarrow q(X) \wedge t(Y), q(X) \vee t(Y) \leftarrow s(X) \wedge s(Y) \wedge \neg p(X,Y), q(X) \leftarrow s(X) \wedge p(X,Y)\}$ and $\{s(X) \leftarrow a(X)\}$.

The SCCs of $G^{+}_{\mathcal{P}}$ are $\{p,q\}$, $\{t\}$, and $\{s\}$. They correspond to the modules $\{p(X,Y) \leftarrow q(X) \wedge t(Y), q(X) \vee t(Y) \leftarrow s(X) \wedge s(Y) \wedge \neg p(X,Y), q(X) \leftarrow s(X) \wedge p(X,Y)\}$, $\{q(X) \vee t(Y) \leftarrow s(X) \wedge s(Y) \wedge \neg p(X,Y)\}$, and $\{s(X) \leftarrow a(X)\}$.

If we denote by C_1 the first SCC of $G^{+}_{\mathcal{P}}$, then the rule $p(X,Y) \leftarrow q(X) \wedge t(Y)$ belongs to $recursive\_rules_{\mathcal{P}}(C_1)$ and the rule $q(X) \vee t(Y) \leftarrow s(X) \wedge s(Y) \wedge \neg p(X,Y)$ belongs to $exit\_rules_{\mathcal{P}}(C_1)$. Moreover, the rule $s(X) \leftarrow a(X)$ is total. $\square$

5 Instantiation Procedure

In this section we present the instantiator module of the DLV system. The main tasks of this module are

- to evaluate total program components (normal stratified components only depending on other normal stratified components and facts), and
- to generate the instantiation of disjunctive or unstratified components.

In order to efficiently evaluate stratified programs (components) we mainly use a technique borrowed from classical deductive databases, an improved version of the generalized semi-naive technique. If the input program is normal and stratified, the instantiator completely evaluates the program; the program has a single stable model, namely the set of the facts in input plus the atoms derived by the instantiation procedure.

If the input program is disjunctive or unstratified, the instantiation procedure cannot completely evaluate the program. However, the optimization technique mentioned above is useful to efficiently generate the instantiation of the non-monotonic part of the program.

In general, two aspects are crucial for the instantiation:

(a) the number of generated ground rules, and

(b) the time needed to generate the grounding.

The size of the grounding generated is important because it strongly influences the computation time of the other modules of the system. A slower instantiation procedure generating a smaller grounding may be preferable to a faster one generating a large grounding, though clearly the time needed by the former can not be ignored and has to be weighed against its benefits.

The main reason $ground(\mathcal{P})$ is often huge compared to $\mathcal{P}$ is that each atom of a rule in $\mathcal{P}$ may be instantiated to many atoms in $B_{\mathcal{P}}$, which leads to combinatorial explosion. However, in a reasonable semantics such as the Stable Model Semantics, most of these atoms may not be derivable whatsoever, and hence such instantiations do not render applicable rules.

Example 5.1 *Consider the following classical deductive database example: Given a parent relationship, find the genealogy tree of each person in the database. Let $\mathcal{P}$ be the program encoding it, where $parent(\_,\_)$ is an input relation:*

$$\begin{array}{rcl} ancestor(X,Y) & \leftarrow & parent(X,Y). \\ ancestor(X,Y) & \leftarrow & ancestor(X,U), ancestor(U,Y). \end{array}$$

Now assume that we have just one fact $parent(thomas, moritz)$ in input. Then $ground(\mathcal{P})$ here consists of $4+8=12$ rules

$$\begin{array}{rcl} ancestor(thomas,thomas) & \leftarrow & parent(thomas,thomas). \\ ancestor(thomas,moritz) & \leftarrow & parent(thomas,moritz). \\ ancestor(moritz,thomas) & \leftarrow & parent(moritz,thomas). \\ ancestor(moritz,moritz) & \leftarrow & parent(moritz,moritz). \\ ancestor(thomas,thomas) & \leftarrow & ancestor(thomas,thomas), \\ & & ancestor(thomas,thomas). \\ ancestor(thomas,moritz) & \leftarrow & ancestor(thomas,moritz), \\ & & ancestor(moritz,moritz). \\ \vdots & & \vdots \\ ancestor(moritz,moritz) & \leftarrow & ancestor(moritz,moritz), \\ & & ancestor(moritz,moritz). \end{array}$$

Concerning issue (a) above, in order to generate the smallest ground program equivalent to the given input program (according to the stable model semantics), we present an algorithm which generates ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$.

5.1 Program Instantiation

Let $\mathcal{P}$ be a non-ground program. Recall that we assume that $\mathcal{P}$ is *safe*, i.e., all variables of a rule r appear in $B^+(r)$. Consequently, in order to instantiate a rule r, we merely have to instantiate $B^+(r)$, which uniquely extends to r. We define the grounding of r w.r.t. a set of ground atoms *NF* $\subseteq B_{\mathcal{P}}$, denoted by $ground(r, \mathit{NF})$, as the set of ground instances r' of r s.t. $B^+(r') \subseteq$ *NF*. The set $ground(r, \mathit{NF})$ is computed by the function *Evaluate*(r, NF) which is described at the end of this section.

The algorithm *Instantiate* is outlined in Figures 3 and 4. It computes a ground program $\Pi \cup$ *T*, where Π is the set of ground rules and *T* is the set of ground atoms derived from $\mathcal{P}$ (i.e., non-disjunctive ground rules with an empty body), which has the same stable models as $\mathcal{P}$.

Furthermore, *Instantiate* computes the set of atoms, denoted by *NF*, which could possibly be derived through the rules of the program and includes only those ground rules which are possibly useful to derive these atoms in Π.

In the following, $EDB_{\mathcal{P}}$ and $IDB_{\mathcal{P}}$ denote the database and intensional part of $\mathcal{P}$, respectively.

Initially, it sets *NF* $= EDB_{\mathcal{P}}$, *T* $= EDB_{\mathcal{P}}$ and $\Pi = \emptyset$. Then, it removes a SCC C from $G_{\mathcal{P}}$ which has no incoming arc (i.e., a source). Consequently, it removes a SCC C' from $G^+_{\mathcal{P}}$ s.t. C' has no incoming arc and $C' \subseteq C$, and generates all instances r' of rules r defining predicates in C' which can possibly derive new atoms, given that the atoms in *NF* are possibly derivable. This is done by calls to *InstantiateRule*.

These rules r' are those rules in $ground(r, \mathit{NF})$ such that every negative total literal in $B^-(r')$ is true w.r.t. *T*. First, we add $H(r')$ to *NF* because each atom in $H(r')$ can possibly be derived. We then remove all positive literals (all negative total literals) in *T* from $B^+(r')$ (from $B^-(r')$). Finally, if the head of r' is disjunction-free and its body became empty after the simplification steps, the head atom is inserted in *T*, otherwise the simplified version of r' is added to Π.

In order to compute such an r', the function *Evaluate* proceeds by matching the atoms in $B^+(r)$ one by one with atoms in *NF* (Δ*NF*) and binding the free variables accordingly in each step, as in the case of a relational join operation. If $r \in exit\_rules_{\mathcal{P}}(C')$, the set Δ*NF* is irrelevant. If r is a linear recursive rule, the semi-naive optimization technique is used and the recursive body atom is matched only with atoms in Δ*NF*; all non-recursive atoms are matched with atoms in *NF*. If r is a non-linear recursive rule, an improved generalized semi-naive technique is used.

5.2 Rule Instantiation

As far as the instantiation itself is concerned, an efficient heuristics is to start with positive literals whose predicate occurs infrequently in *NF* (Δ*NF*) and whose variables we find in most different body literals. Therefore, before starting the matching of the atoms in $B^+(r)$ one by one, we first order the positive literals of the body by the increasing cardinality of their ground occurrences in *NF* (Δ*NF*) and by the decreasing number of their common variables. The positive literals whose variables are unique, are placed at the end of the re-tabulated rule body even if the cardinalities of their ground occurrences in *NF* (Δ*NF*) are small. The reason is that in this case the join operation with the rest of the body literals is equivalent to the cartesian product.

We describe next how the function *Evaluate* proceeds when r is an exit rule or a linear recursive rule. The case when r is a non-linear recursive rule is more complicated and we will describe it by giving an example.

At the i-th step, all literals L_j, $1 \leq j < i$, have been matched and we try to match the i-th body literal L_i. Note that some variables of L_i could already be bounded due to the previous steps. There are two possibilities: (i) L_i can be matched with some atom in *NF* (if $L_i \notin C'$) or in Δ*NF* (if $L_i \in C'$). If L_i is not the last body literal, we compute the matching of L_i and try to match the literal L_{i+1}. If L_i is the last body literal, we add the new ground instance of r to $ground(r, \mathit{NF})$ and try to match L_i with another atom. (ii) L_i can not be matched with any atom in *NF* (if $L_i \notin C'$) or in Δ*NF* (if $L_i \in C'$). If L_i is the first body literal ($i = 1$), no further ground instance of r can be derived and the function *Evaluate* exits and returns the set $ground(r, \mathit{NF})$. If $i > 1$, we backtrack to the previous literal L_{i-1} and try to match it with another atom in *NF* (if $L_{i-1} \notin C'$) or in Δ*NF* (if $L_{i-1} \in C'$).

If r is a non-linear recursive rule, the function *Evaluate* proceeds in a similar way. We need to mark one recursive body literal at a time. Each time the matching of the first body literal fails (in the case of the exit rule or linear recursive rule, the function would exit and return the set $ground(r, \mathit{NF})$), we unmark the current marked recursive body literal, mark the next recursive body literal and the same steps as in the case of exit rule or linear recursive rule are followed, with some differences: (i) the marked recursive body literal can be matched only with atoms in Δ*NF*, (ii) the recursive body literals to the left of the marked recursive body literal can be matched only with atoms in *NF* $-$ Δ*NF*, and (iii) the recursive body literals to the right of the marked recursive body literal can be matched only with atoms in *NF*. The classical generalized semi-naive technique makes no difference between the recursive body literals laying to the left or right side of the marked recursive body literal and it therefore generates duplicated ground instances. Our improvement avoids generating the same ground rule more than once. The function *Evaluate* exits only when the matching of the first body literal fails and there is no other recursive body literal to be marked.

For efficiency reasons, first non-recursive rules are instantiated once and for all. Then, the recursive rules are repeatedly instantiated until *NF* remains unchanged.

After all unmarked rules defining the predicates from C' have been instantiated, they are marked in order to avoid grounding them again when predicates from their heads belong to other SCCs. Consequently, the predicates from C' are removed from C.

After that, another SCC source from $G^+_{\mathcal{P}}$ included in C is processed and removed from C until C becomes empty. When C becomes empty, the SCCs of $G^+_{\mathcal{P}}$ included in the next source from $G_{\mathcal{P}}$ are processed.

For instance, in the example, the unique source $C_1 = \{s\}$ of $G_{\mathcal{P}}$ is taken first. Obviously, the only source of $G^+_{\mathcal{P}}$ included in C_1 is $\{s\}$ itself, and it is therefore processed. Once $\{s\}$ has been removed from $G_{\mathcal{P}}$, $C_2 = \{p, q, t\}$ becomes the (unique) source of $G_{\mathcal{P}}$ and is therefore taken. The only SCC source of $G^+_{\mathcal{P}}$ contained in C_2 is $C'_1 = \{t\}$ which is processed and removed from $G^+_{\mathcal{P}}$. Thus, $C'_2 = \{p, q\}$ becomes a source and is processed at last, completing the instantiation process.

Each time we pick up a source C' from $G^+_{\mathcal{P}}$ for processing, all possible derivable ground instances of C' are generated once and for all by using the ground instances of the sources processed

```
Procedure Instantiate( $\mathcal{P}$: SafeProgram;
                       $G_{\mathcal{P}}$: dependency graph;
                       var $\Pi$: GroundProgram;
                       var $T$: SetOfAtoms)
var
    $C$, $C'$: SetOfPredicates;
    $NF$, $NF1$, $\Delta NF$: SetOfAtoms;
begin
    $NF := EDB_{\mathcal{P}}$; $T := EDB_{\mathcal{P}}$; $\Pi := \emptyset$;
    while $G_{\mathcal{P}} \neq \emptyset$ do
        Remove a SCC $C$ from $G_{\mathcal{P}}$ without incoming edges;
        while $C \neq \emptyset$ do
            Remove a SCC $C'$ from $G^{+}_{\mathcal{P}}$ without incoming edges s.t. $C' \subseteq C$;
            $NF1 := NF$;
            for each unmarked rule $r \in exit\_rules_{\mathcal{P}}(C')$ do
                InstantiateRule($\mathcal{P}, r, \emptyset, NF, T, \Pi$);
            $\Delta NF := NF - NF1$;
            repeat
                $NF1 := NF$;
                for each unmarked rule $r \in recursive\_rules_{\mathcal{P}}(C')$ do
                    InstantiateRule($\mathcal{P}, r, \Delta NF, NF, T, \Pi$);
                $\Delta NF := NF - NF1$;
            until $\Delta NF = \emptyset$
            Mark all unmarked rules from $exit\_rules_{\mathcal{P}}(C') \cup recursive\_rules_{\mathcal{P}}(C')$;
            $C := C \setminus C'$;
        end while
    end while
end function;
```

Figure 3: Computation of the (simplified) instantiated program

previously. In this way we optimize (a), i.e., we generate only ground rules whose head contains atoms which can possibly be derived from $\mathcal{P}$.

Note that if $\mathcal{P}$ is a normal (disjunction-free) stratified program, the grounding is empty because the body of all grounded rules is empty and their head atom is added to *T*.

Example 5.2 *Reconsider $\mathcal{P}$ from Example 4.1, and assume $EDB_{\mathcal{P}} = \{a(2)\}$. Then,* Instantiate *computes the following ground program Π of $\mathcal{P}$:*

$$\begin{array}{rclrcl} p(1,2) \vee p(2,3) & \leftarrow & & q(1) \vee q(3) & \leftarrow & p(1,2), p(2,3), \neg t(2) \\ t(2) & \leftarrow & & t(3) & \leftarrow & q(3), p(2,3) \end{array}$$

Evaluation of node $\{p\}$ yields the upper left rule of Π, *and* $\mathrm{NF} = \{a(2), p(1,2), p(2,3)\}$. *We then evaluate the node $\{q\}$ and get the upper right rule of* Π, *while* NF *becomes* $\{a(2), p(1,2), p(2,3),$

```
**Procedure** *InstantiateRule*( $\mathcal{P}$: SafeProgram;
                                 $r$: Rule;
                                 $\Delta$*NF*: SetOfAtoms;
                                 **var** *NF*, *T*: SetOfAtoms;
                                 **var** $\Pi$: GroundProgram )
**var**
    $H$ : SetOfAtoms;
    $B^+, B^-$: SetOfLiterals;
**begin**
    **for each** instance $H \leftarrow B^+, B^-$ of $r$ in $Evaluate(r, \Delta NF, NF)$ **do**
        **if** $(\neg.B^- \cap T = \emptyset) \wedge (H \cap T = \emptyset)$ **then**
            $NF := NF \cup H$;
            Remove all positive literals in *T* from $B^+$;
            Remove all negative total literals in *T* from $B^-$;
            **if** $(B^+ = \emptyset) \wedge (|H| = 1)$ **then**
                $T := T \cup H$
            **else**
                $\Pi := \Pi \cup \{H \leftarrow B^+, B^-\}$
            **end if**
        **end if**
**end procedure**;
```

Figure 4: Instantiation of a single rule

$q(1), q(3)\}$. *Finally, we consider the node* $\{t\}$. *The rule* $t(X) \leftarrow a(X)$ *yields* $t(2) \leftarrow$ *and the rule* $t(X) \leftarrow q(X), p(Y, X)$ *yields* $t(3) \leftarrow q(3), p(2, 3)$.

Note that $ground(\mathcal{P})$ *contains 1+3+27+9=40 rules, while* Instantiate *generates only 4 rules.*

Theorem 5.1 *Let* $\mathcal{P}$ *be a safe disjunctive datalog program, and* $\Pi \cup \mathrm{T}$ *be the ground program generated by* Instantiate*(*$\mathcal{P}$*). Then*

1. $\mathcal{P}$ *and* $\Pi \cup \mathrm{T}$ *have the same stable models;*

2. *if* $\mathcal{P}$ *is a normal stratified program then* $\Pi = \emptyset$ *and* T *is the single stable model of* $\mathcal{P}$.

Proof. The thesis trivially follows from the fact that *NF* contains all the ground atoms which can be possibly derived from the original program, and that $NF = T$ in case of normal and stratified programs. We will show this by induction on the components of $\mathcal{P}$.

Let C_1 be a SCC component of $G_{\mathcal{P}}$ without incoming edges, $\mathcal{P}_1$ be the corresponding module of $\mathcal{P}$, and *NF*$_1$, T_1 be the sets NF,T, respectively, computed by the algorithm after the evaluation of $\mathcal{P}_1$. For the evaluation of each single component, the algorithm evidently exploits a generalization of the seminaïve algorithm [Ull88]. In absence of sources of non-determinism (disjunction/non-stratified negation), such generalization behaves exactly as the standard one, computing a set of atoms (T_1) which coincides with the (unique) stable model of the module; no ground rule is produced. In presence of some source of non-determinism (disjunctive/non-stratified rules), the algorithm behaves as before while processing the "deterministic" rules, while it simply generates the ground instances of the disjunctive/non-stratified ones, dropping those which are trivially satisfied (i.e., bodies with no chance to be true). It is worth noting that both the atoms added to T_1 and those defined by the generated ground rules are added to NF_1. Thus, this set will clearly contain all atoms which can be possibly derived from the original program.

When a generic component C_i of $G_{\mathcal{P}}$ is going to be evaluate, it does not have any incoming edges, either because it does not depend on any other component, or because all the components it depends on have been already processed (and thus removed from $G_{\mathcal{P}}$). Let NF_{i-1} and T_{i-1} be the sets computed by the algorithm after the evaluation of component C_{i-1}, and let us assume that NF_{i-1} contains all atoms which can be possibly derived from previous modules, and that T_{i-1} contains the atoms defined by previous modules that can be recognized as true in all possible stable models. We next show that NF_i (i.e., the set NF computed by the algorithm after the evaluation of component C_i) contains all atoms which can be possibly derived from previous modules plus P_i. Indeed, if it was not the case, then there should be a ground atom which does not appear in NF_i and may be derived by some rule of P_i. But NF_i has been computed, starting from atoms in NF_{i-1}, by means of an algorithm which, similarly as stated above, evidently computes all the ground atoms which may possibly be true. This means that such "non-computed" atom should have been missed (at some step) while evaluating $NFi - 1$, but this cannot be the case because of the inductive hypothesis.

It is easy to see that, if all the components $C_1 \dots C_i$ are normal and stratified, $NF_{i-1} = T_{i-1}$, and also after the evaluation of C_i, $NF_i = T_i$.

6 Rewriter

As we have seen, both the size of the ground instantiation generated and the time taken for the instantiation are important factors for the quality of a instantiator for disjunctive databases.

In the following, we propose a further optimization technique that descends from query optimization techniques in the field of relational databases where the input is rewritten to avoid the generation of redundant ground rules, which often results in a smaller and faster instantiation at the same time.

To give an intuition, consider the rule r_1:

$$p(X) \leftarrow r(X, Y, Z), q(Z, V, S), V < S.$$

Deductive database systems, based on a bottom-up computational model, evaluate the relational algebra expression corresponding to the body of the rule and add the result of the evaluation to a relation corresponding to the head predicate [Ull89], and as we have seen also DLV proceeds exactly this way if r and q are either base predicates or predicates defined by a normal stratified (sub)program. Otherwise, a similar process is used to generate the ground instances of the rule.

The relational algebra expression corresponding to the above rule is

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

where R and Q are the relations corresponding to predicates 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, most relational database systems will 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 the following, 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 these techniques for the evaluation of traditional, non-disjunctive stratified deductive databases is well known and has already been implemented in several deductive database systems [Ull89]. 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 example above are defined by disjunctive rules. If the Intelligent Grounding of DLV has to generate the ground instances of r_1, using the technique shown above, it first rewrites r_1 as follows

$$\begin{array}{l} p(X) \leftarrow r'(X, Z), q'(Z). \\ r'(X, Z) \leftarrow r(X, Y, Z). \\ q'(Z) \leftarrow q(Z, V, S), V < S. \end{array}$$

and then instantiates the rewritten program, which has exactly the same stable models as the original one modulo the primed predicates r' and q' (which DLV marks as internal and omits from its output).

The main advantages we obtain are

- the speed-up of the evaluation of normal stratified programs (which are completely solved by DLV's Intelligent Grounding such that the ground instantiation is not materialized at all in this case);
- the speed-up of the instantiation process for general (disjunctive or unstratified) programs, and, most importantly,
- the drastic reduction of the size of the ground instantiation (for general programs), which dramatically improves overall system performance, as pointed out in Section 7.

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

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

Basic Case

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. As we have seen in Section 5, the instantiation 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. We can thus eliminate X_i by projecting p on all variables $X_k, k \neq i$, obtaining an auxiliary predicate p', substitute p by p' in the body of r, and add a new (non-ground) rule

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

In this way the generation of ground instances of r which differ only on the binding of X_i is avoided.

General Case: Groups of Atoms

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

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

$$a(X) \vee b(Y) \leftarrow c(X, Z, W), d(Z, Y), e(Y, W).$$

where the variable Z appears in both c and d but not in e, a or b. Here we add the rule

$$f(X,Y,W) \leftarrow c(X,Z,W), d(Z,Y).$$

and substitute r_2 by a new rule r_2',

$$a(X) \vee b(Y) \leftarrow f(X,Y,W), e(Y,W).$$

Again, the ground instances for r_2' are generated faster and will be smaller than the one of r_2 as well, because the generation of useless ground instances is avoided by eliminating Z from the rule body.

Exploiting Built-In Predicates

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

$$p(X,Y,Z) \leftarrow ..., q(X,Y,Z), ..., X < Y, ...$$

The set of ground instances of q can be narrowed, possibly sensibly, before starting to instantiate r_3 by adding a new rule r_3',

$$q'(X,Y,Z) \leftarrow q(X,Y,Z), X < Y.$$

and replacing r_3 by

$$p(X,Y,Z) \leftarrow ..., q'(X,Y,Z), ...$$

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 $X < Y$.

Similar to the base case, a 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,

$$a(X) \vee b(Y) \leftarrow c(X,Z), d(V,W), e(Y,V,W), Z < W.$$

where we can add the rule

$$f(X,V,W) \leftarrow c(X,Z), d(V,W), Z < W.$$

and substitute r_4 by the new rule r_4',

$$a(X) \vee b(Y) \leftarrow f(X,V,W), e(Y,V,W).$$

This way we avoid the situation where, 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.

Let $\mathcal{P}$ be a (non-ground) program and $\mathcal{P}'$ the program obtained from $\mathcal{P}$ by applying the rewriting optimization technique described in this section. Given a stable model M' for $\mathcal{P}'$, $\mathcal{P}(M')$ is the set of literals obtained from M' by eliminating all the auxiliary literals, i.e., $\mathcal{P}(M')$ 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 stable models of $\mathcal{P}$. The result supporting the above statement is the following:

Theorem 6.1 *For each stable model M' for $\mathcal{P}'$, $\mathcal{P}(M')$ is a stable model for $\mathcal{P}$. Moreover, for each stable model M for $\mathcal{P}$ there exists a stable model M' for $\mathcal{P}'$ such that $\mathcal{P}(M') = M$.*

7 Experimental Results

In order to check the efficiency of the rewriting technique, we have implemented it in the grounding engine of DLV, and we have run it on a collection of benchmark programs taken from different domains.

We provide below a very short description of the problems which are encoded in the benchmark programs.

7.1 Benchmark Programs

CONSTRAINT-3COL 3col, constraint-satisfaction-like encoding, on a graph with 30 nodes and 40 edges.

CRISTAL A deductive databases application developed at CERN in Switzerland involving complex knowledge manipulations on databases, .

DECOMP Decide whether a conjunctive query has hypertree width at most K [GLS99].

HANOI ”Towers of Hanoi”with 3 stacks, 4 disks, and 15 steps.

TIMETABLING A timetable problem for the first year of the faculty of Science of the University of Calabria.

BLOCKSWORLD A typical planning problem where some blocks, placed on a table, have to be moved from an initial position to a desired final position.

7.2 Experimental results and discussion

We implemented in DLV the technique described in Section 6 and we tested it by using the above benchmark problems. All experiments were performed on a machine equipped with Pentium Intel 4, 1400 MHz, 256MB of main memory. The binaries were produced with GCC 2.95.2.

The results of our tests are shown in Table 1.

There, the first column describes the benchmark program; columns 2-3 (resp.3-4) refer to DLV without (resp. with) the rewriting technique and report the size (number of rules) of the output

	DLV		DLV + Rewriter	
Problem	size	time	size	time
CONSTRAINT-3COL	16394496	325.63	512330	23.80
CRISTAL	0	10.21	0	9.93
DECOMP	922	24.17	922	22.72
HANOI	68720	2.08	12110	0.51
TIMETABLING	557814	248.95	194247	195.50
BLOCKSWORLD_1	447004	15.83	16906	1.26
BLOCKSWORLD_2	517192	19.43	17179	1.20

Table 1: Instantiation times of DLV without resp. with the rewriter technique (times in seconds)

instantiation and the time (in seconds) taken to generate it. For normal stratified programs DLV does not produce any instantiation but outputs the single stable model; thus the size reported in this case is 0.

It is evident that the proposed optimization considerably improves the performance of the instantiator. In particular, it provides a tremendous performance boost in many cases (up to 94%). Notably, it also allows to significantly reduce the size of the ground program which is crucial for the performances of the other modules of the DDDBs. Indeed, we are currently carrying out some other experiments in order to evaluate the impact of the reduced size of the ground program on the modules which are in charge to compute the stable models; preliminary results confirm the intuition that the smaller the size of the instantiation, the faster the stable model computation.

8 Conclusions

We described the instantiation procedure of the DLV system and we proposed an optimization technique descending from query optimization techniques, integrated it into the system and carried out an experimental analysis. The proposed technique applies to both rules and constraints (as the latter are just rules with an empty head).

The results confirm that the main advantages of the proposed optimization technique are

(a) the speed-up of the evaluation of normal stratified programs,

(b) the speed-up of the instantiation process for general (disjunctive or unstratified) programs, and

(c) the drastic reduction of the size of the ground instantiation (in general).

References

[AB94] K. Apt and N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[BD95] Stefan Brass and Jürgen Dix. Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation. In Leon Sterling, editor, *Proceedings of the 12th Int. Conf. on Logic Programming*, pages 199–213, Tokyo, June 1995. MIT Press.

[CCCR$^+$90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm. In *Proceedings of 1990 ACM-SIGMOD International Conference*, pages 225–236, Atlantic City, NJ, May 1990.

[CGK$^+$90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[Dix95] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pages 241–329. DeGruyter, 1995.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[ELM$^+$97] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A Deductive System for Nonmonotonic Reasoning. In Jürgen Dix and Ulrich Furbach and Anil Nerode, editor, *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, Dagstuhl, Germany, July 1997. Springer.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems – PODS'99*, pages 21–32, May 31st – June 2nd 1999. Full paper in *Journal of Computer and System Sciences*.

[LMR92] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[LR95] Nicola Leone and Pasquale Rullo. BQM: A System Integrating Logic, Objects, and Non-Monotonic Reasoning. In *Invited Paper on 7th IEEE International Conference on Tools with Artificial Intelligence*, Washington, November 1995.

[LT94] V. Lifschitz and H. Turner. Splitting a Logic Program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[Min82] Jack Minker. On Indefinite Data Bases and the Closed World Assumption. In D.W. Loveland, editor, *Proceedings* 6^{th} *Conference on Automated Deduction (CADE '82)*, number 138 in Lecture Notes in Computer Science, pages 292–308, New York, 1982. Springer.

[PDR91] G. Phipps, M. A. Derr, and K.A. Ross. Glue-NAIL!: A Deductive Database System. In *Proceedings ACM-SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[Prz90] T. Przymusinski. Stationary Semantics for Disjunctive Logic Programs and Deductive Databases. In *Proceedings of North American Conference on Logic Programming*, pages 40–62, 1990.

[Prz91] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[Prz95] T. Przymusinski. Static Semantics for Normal and Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 14:323–357, 1995.

[Ros90] K.A. Ross. The Well-Founded Semantics for Disjunctive Logic Programs. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 385–402. Elsevier Science Publishers B. V., 1990.

[RSS92] R. Ramakrishnan, D. Srivastava, and S Sudarshan. CORAL – Control, Relations and Logic. In *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia, Canada, 1992.

[Sak89] C. Sakama. Possible Model Semantics for Disjunctive Databases. In *Proceedings First Intl. Conf. on Deductive and Object-Oriented Databases (DOOD-89)*, pages 369–383, Kyoto, Japan, 1989. North-Holland.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Management System*. New York: Academic, 1988.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.