TLP **8** (2): 129–165, 2008.
doi:10.1017/S1471068407003158 First published online 21 June 2007 Printed in the United Kingdom

Experimenting with recursive queries in database and logic programming systems

G. TERRACINA, N. LEONE, V. LIO and C. PANETTA

Dipartimento di Matematica, Università della Calabria,
Via P. Bucci, 87030, Rende (CS), Italy
(*e-mail:* {terracina, leone, lio, panetta}@mat.unical.it)

submitted 16 October 2006; revised 12 March 2007; accepted 20 April 2007

Abstract

This article considers the problem of reasoning on massive amounts of (possibly distributed) data. Presently, existing proposals show some limitations: (i) the quantity of data that can be handled contemporarily is limited, because reasoning is generally carried out in main-memory; (ii) the interaction with external (and independent) Database Management Systems is not trivial and, in several cases, not allowed at all; and (iii) the efficiency of present implementations is still not sufficient for their utilization in complex reasoning tasks involving massive amounts of data. This article provides a contribution in this setting; it presents a new system, called DLV^{DB}, which aims to solve these problems. Moreover, it reports the results of a thorough experimental analysis we have carried out for comparing our system with several state-of-the-art systems (both logic and databases) on some classical deductive problems; the other tested systems are LDL++, XSB, Smodels, and three top-level commercial Database Management Systems. DLV^{DB} significantly outperforms even the commercial database systems on recursive queries.

KEYWORDS: Deductive Database Systems, Answer Set Programming/Declarative Logic Programming, recursive queries, benchmarks

1 Introduction

The problem of handling massive amounts of data received much attention in the research related to the development of efficient Database Management Systems (DBMSs). In this scenario, a mounting wave of data-intensive and knowledge-based applications such as Data Mining, Data Warehousing, and Online Analytical Processing has created a strong demand for more powerful database languages and systems. An important effort in this direction has been carried out with the introduction of the latest standard for SQL, namely, SQL99 (SQL: 1999) (American National Standards Institute 1999), which provides, among other features, support to object-oriented databases and recursive queries.

However, the adoption of SQL99 is still far from being a "standard"; in fact, almost all current DBMSs do not fully support SQL99 and, in some cases, they

adopt proprietary (nonstandard) language constructs and functions to implement parts of it. Moreover, the efficiency of current implementations of SQL99 constructs and their expressiveness are still not sufficient for performing complex reasoning tasks on huge amounts of data.

The needed expressiveness for reasoning tasks can be provided by logic-based systems. In fact, declarative logic programming provides a powerful formalism capable of easily modeling and solving complex problems. The recent development of efficient logic-based systems such as DLV (Leone *et al.* 2006), Smodels (Niemelä *et al.* 2000), XSB (Rao *et al.* 1997), ASSAT (Lin and Zhao 2002, 2004), Cmodels (Giunchiglia *et al.* 2004, 2006), CLASP (Gebser *et al.* 2007), etc. has renewed the interest in the area of non-monotonic reasoning and declarative logic programming for solving real-world problems in a number of application areas. However, "data-intensive" problems cannot be handled in a typical logic programming system working in main-memory.

In the past, Deductive Database Systems (DDSs) have been proposed to combine the expressive power of logic-based systems with the efficient data management of DBMSs (Gallaire *et al.* 1984; Ceri *et al.* 1990; Grant and Minker 1992; Arni *et al.* 2003); basically, they are an attempt to adapt typical Datalog systems, which have a "smalldata" view of the world, to a "largedata" view of the world via intelligent interactions with some DBMSs. Recently, emerging application contexts such as the ones arising from the natural recursion across nodes in the Internet, or from the success of intrinsically recursive languages such as XML (Winslett 2006), renewed the interest in such kinds of systems (Abiteboul *et al.* 2005; Loo *et al.* 2005).

However, the main limitations of currently existing DDSs reside both in the fact that reasoning is still carried out in main-memory—this limits the amount of data that can be handled—and in the limited interoperability with generic, external, DBMSs they provide. In fact, generally, the reasoning capabilities of these systems are tailored on a specific (either commercial or ad hoc) DBMS.

Summarizing: (i) Database systems are nowadays robust and flexible enough to efficiently handle large, possibly distributed, amounts of data; however, their query languages are not sufficiently expressive to support reasoning tasks on such data. (ii) Logic-based systems are endowed with highly expressive languages, allowing them to support complex reasoning tasks, but they work in main-memory, and, hence, can only handle limited amounts of data. (iii) DDSs allow to access and manage data stored in DBMSs, however, they perform their computations mainly in main-memory and provide limited interoperability with external (and possibly distributed) DBMSs.

This work provides a contribution in this setting, bridging the gap between logic-based DDSs and DBMSs. It presents a new system, named DLV^{DB}, which is logic-based (like a DDS) but can do all the work in mass-memory and, in practice, does not have any limitation in the dimension of input data; moreover, it is capable to exploit optimization techniques both from DBMS (e.g., join orderings, Garcia-Molina *et al.* 2000) and DDS theory (e.g., magic sets, Beeri and Ramakrisnhan 1991; Mumick *et al.* 1996).

DLV^{DB} allows for two typologies of execution: (i) direct database execution, which evaluates logic programs directly on database, with a very limited usage

of main-memory but with some limitations on the expressiveness of the queries, and (ii) main-memory execution, which loads input data from different (possibly distributed) databases and executes the logic program directly in main-memory. In both cases, interoperation with databases is provided by ODBC connections; these allow handling, in a quite simple way, data residing on various databases over the network. To avoid possible confusion, in the following we use the symbol DLV^{DB} to indicate the whole system when the discussion is independent of the execution modality; however, when it is needed to distinguish between the two execution modalities, we use the symbol DLV^{IO} to indicate the main-memory execution, whereas the symbol DLV^{DB} to indicate the direct database execution.

Summarizing, the overall contributions of this work are the following: (i) The development of a full-fledged system enhancing in different ways the interactions between logic-based systems and DBMSs. (ii) The development of an efficient, purely database-oriented, evaluation strategy for logic programs that minimizes the usage of main-memory and maximizes the advantages of optimization techniques implemented in existing DBMSs. (iii) The definition of a framework for carrying out an experimental comparative analysis of the performance of state-of-the-art systems and DLV^{DB}. (iv) The execution of a thorough experimentation that shows that DLV^{DB} beats, often with orders of magnitude, logic-based systems (LDL++, XSB, and Smodels[1]) and even commercial DBMSs both for running times and amount of handled data on classical deductive problems (Bancilhon and Ramakrishnan 1988).

The work is organized as follows. Section 2 presents the reasoning language supported by the system, whereas Section 3 describes the functionalities it provides. In Section 4, the main implementation principles adopted in the development of DLV^{DB} are discussed, and Section 5 illustrates its general architecture. Section 6 first presents an overview of the state-of-the-art systems related to DLV^{DB}, then it describes the experimental analysis we have carried out to compare DLV^{DB} with these systems on classical DDS problems. Finally, in Section 7, we draw our conclusions.

2 The reasoning language of the system

In this section, we briefly describe the syntax and the semantics of the reasoning language adopted by the DLV^{DB} system. This is basically Disjunctive Logic Programming (DLP) with aggregate functions under the answer set semantics; we refer to this language as $\text{DLP}^{\mathcal{A}}$ in the following. The interested reader can find all details about $\text{DLP}^{\mathcal{A}}$ in Faber *et al.* (2004).

Before starting the presentation, it is worth pointing out that the direct database execution modality supports only a strict subset of the reasoning language supported by the main-memory execution. In particular, while DLV^{IO} supports the whole language of DLV (including disjunction, unlimited negation, and stratified aggregates), DLV^{DB} supports or-free programs with stratified negation and aggregates.

[1] It is worthwhile noting that, since benchmark programs are stratified, they are completely solved by the grounding layer of Smodels (LParse). This is the reason why we have not experimented with ASSAT, Cmodels, and CLASP, as they also use LParse for grounding.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

2.1 Syntax

We assume that the reader is familiar with standard DLP; we refer to atoms, literals, rules, and programs of DLP as *standard atoms, standard literals, standard rules*, and *standard programs,* respectively. For further background, see Baral (2002) and Gelfond and Lifschitz (1991).

2.1.1 Set terms

A (DLP$^{\mathcal{A}}$) *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : Conj\}$, where *Vars* is a list of variables and *Conj* is a conjunction of standard atoms.[2] A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where $\bar{t}$ is a list of constants and *Conj* is a ground (variable free) conjunction of standard atoms.

2.1.2 Aggregate functions

An *aggregate function* is of the form $f(S)$, where S is a set term and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

The aggregate functions that are currently supported are `#count` (number of terms), `#sum` (sum of non-negative rational numbers), `#min` (minimum term, undefined for empty set), `#max` (maximum term, undefined for empty set), and `#avg` (average of non-negative rational numbers).[3]

2.1.3 Aggregate literals

An *aggregate atom* is $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leqslant, >, \geqslant\}$ is a predefined comparison operator, and T is a term (variable or constant) referred to as guard.

An example of aggregate atom is: $\#\mathtt{max}\{Z : r(Z), a(Z, V)\} > Y$.

An *atom* is either a standard (DLP) atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol `not`; if A is an aggregate atom, then L is an *aggregate literal.*

2.1.4 DLP$^{\mathcal{A}}$ programs

A DLP$^{\mathcal{A}}$ *rule* r is a construct

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m,$$

where $a_1, \ldots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, $n \geqslant 0$, and $m \geqslant k \geqslant 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is referred to as the *head* of r, whereas the conjunction $b_1, \ldots, b_k,$ not $b_{k+1}, \ldots,$ not b_m is the *body* of r. We denote the set $\{a_1, \ldots, a_n\}$ of the

[2] Intuitively, a symbolic set $\{X : a(X, Y), p(Y)\}$ stands for the set of X-values making $a(X, Y), p(Y)$ true, that is, $\{X \mid \exists Y\, s.t.\ a(X, Y), p(Y)\ is\ true\}$.

[3] The first two aggregates correspond respectively to the cardinality and weight constraint literals of Smodels.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

head atoms by $H(r)$ and the set $\{b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{not } b_m\}$ of the body literals by $B(r)$. $B^+(r)$ and $B^-(r)$ denote respectively the set of positive literals and the set of negative literals occurring in $B(r)$.

A DLP$^{\mathscr{A}}$ *program* $\mathscr{P}$ is a set of DLP$^{\mathscr{A}}$ rules.

In addition, DLP$^{\mathscr{A}}$ allows for built-in predicates (Faber and Pfeifer, 1996) in its rules, such as the comparative predicates equality, less than, and greater than (=, <, >) and arithmetic predicates such as addition or multiplication (+, *).

2.1.5 Safety

A *global* variable of a rule r is a variable appearing in a standard atom of r; all other variables are *local* variables. A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard literal in the body of r; (ii) each local variable of r appearing in a symbolic set $\{Vars : Conj\}$ appears in an atom of $Conj$, and (iii) each guard of an aggregate atom of r is a constant or a global variable. A program $\mathscr{P}$ is safe if all $r \in \mathscr{P}$ are safe. In the following, we assume that DLP$^{\mathscr{A}}$ programs are safe.

Let the *level mapping* of a program $\mathscr{P}$ be a function $||\ ||$ from the predicates in $\mathscr{P}$ to finite ordinals; moreover, given an atom $A = p(t_1, \ldots, t_n)$, we denote by $Pred(A)$ its predicate p.

2.1.6 Stratified^{not} programs

A DLP$^{\mathscr{A}}$ *program* $\mathscr{P}$ is called *stratified*^{not} (Apt *et al.* 1988; Przymusinski 1988), if there is a level mapping $||\ ||_s$ of $\mathscr{P}$ such that, for every rule r: (1) for any $l \in B^+(r)$, and for any $l' \in H(r)$, $||Pred(l)||_s \leqslant ||Pred(l')||_s$; (2) for any $l \in B^-(r)$, and for any $l' \in H(r)$, $||Pred(l)||_s < ||Pred(l')||_s$; and (3) for any $l, l' \in H(r)$, $||Pred(l)||_s = ||Pred(l')||_s$.

2.1.7 Stratified^{aggr} programs

A DLP$^{\mathscr{A}}$ *program* $\mathscr{P}$ is called *stratified*^{aggr} (Dell'Armi *et al.* 2003b), if there is a level mapping $||\ ||_a$ of $\mathscr{P}$ such that, for every rule r: (1) if l appears in the head of r, and l' appears in an aggregate atom in the body of r, then $||Pred(l')||_a < ||Pred(l)||_a$; and (2) if l appears in the head of r, and l' occurs in a standard atom in the body of r, then $||Pred(l')||_a \leqslant ||Pred(l)||_a$. (3) If both l and l' appear in the head of r, then $||Pred(l')||_a = ||Pred(l)||_a$.

Example 2.1
Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:

$$q(X) \text{ :- } p(X), \#\texttt{count}\{Y : a(Y, X), b(X)\} \leqslant 2. \qquad p(X) \text{ :- } q(X), b(X).$$

The program is stratified^{aggr}, as the level mapping $||a|| = ||b|| = 1$, $||p|| = ||q|| = 2$ satisfies the required conditions. If we add the rule $b(X) \text{ :- } p(X)$, then no level-mapping exists and the program becomes not stratified^{aggr}. □

Intuitively, the property stratified^{aggr} forbids recursion through aggregates.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

2.1.8 Supported languages

The direct database execution modality (DLV^{DB}) currently supports only $\text{DLP}^{\mathcal{A}}$ programs that are disjunction free, stratified^{not}, and strat-ified^{aggr}. Note that both built-in predicates and aggregates are supported.

Conversely, the main-memory execution modality (DLV^{IO}) supports all $\text{DLP}^{\mathcal{A}}$ programs that are stratified^{aggr}. As a consequence, unrestricted negation, disjunction, and non-recursive aggregates are supported.

2.2 Answer set semantics

2.2.1 Universe and base

Given a $\text{DLP}^{\mathcal{A}}$ program $\mathscr{P}$, let $U_{\mathscr{P}}$ denote the set of constants appearing in $\mathscr{P}$, and $B_{\mathscr{P}}$ be the set of standard atoms constructible from the (standard) predicates of $\mathscr{P}$ with constants in $U_{\mathscr{P}}$. Given a set X, let $\overline{2}^{X}$ denote the set of all multisets over elements from X. Without loss of generality, we assume that aggregate functions map to Q (the set of rational numbers).

2.2.2 Instantiation

A *substitution* is a mapping from a set of variables to $U_{\mathscr{P}}$. A substitution from the set of global variables of a rule r (to $U_{\mathscr{P}}$) is a *global substitution for* r; a substitution from the set of local variables of a symbolic set S (to $U_{\mathscr{P}}$) is a *local substitution for* S. Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of* S is the following ground set of pairs $inst(S)$: $\{\langle \gamma(Vars) : \gamma(Conj)\rangle \mid \gamma \text{ is a local substitution for } S\}$.[4]

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathscr{P})$ of a program $\mathscr{P}$ is the set of all possible instances of the rules of $\mathscr{P}$.

Example 2.2
Consider the following program $\mathscr{P}_1$:

$$q(1) \vee p(2,2). \qquad q(2) \vee p(2,1). \qquad t(X) :\!- q(X), \#\mathtt{sum}\{Y : p(X,Y)\} > 1.$$

The instantiation $Ground(\mathscr{P}_1)$ is the following:

$$\begin{array}{ll} q(1) \vee p(2,2). & t(1) :\!- q(1), \#\mathtt{sum}\{\langle 1 : p(1,1)\rangle, \langle 2 : p(1,2)\rangle\} > 1. \\ q(2) \vee p(2,1). & t(2) :\!- q(2), \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1. \end{array}$$

□

2.2.3 Interpretations

An *interpretation* for a $\text{DLP}^{\mathcal{A}}$ program $\mathscr{P}$ is a set of standard ground atoms, that is, $I \subseteq B_{\mathscr{P}}$. A positive literal A is true w.r.t. I, if $A \in I$; it is false otherwise. A negative literal not A is true w.r.t. I; if $A \notin I$, it is false otherwise.

An interpretation also provides a meaning for aggregate literals.

[4] Given a substitution σ and a $\text{DLP}^{\mathcal{A}}$ object Obj (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

Let I be an interpretation. A standard ground conjunction is true (resp. false) w.r.t. I if all its literals are true. The meaning of a set, an aggregate function, and an aggregate atom under an interpretation is a multiset, a value, and a truth value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset of the first constant of the elements in S whose conjunction is true w.r.t. I. More precisely, let $I(S)$ denote the multiset $[t_1 \mid \langle t_1, \ldots, t_n : Conj \rangle \in S \wedge\ Conj$ *is true w.r.t.* I]. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathscr{P}$).

An instantiated aggregate atom $A = f(S) \prec k$ is *true w.r.t.* I if: (i) $I(f(S)) \neq \bot$, and (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal not $A = \text{not}(f(S) \prec k)$ is *true w.r.t.* I if: (i) $I(f(S)) \neq \bot$, and (ii) $I(f(S)) \prec k$ does not hold; otherwise, A is false.

2.2.4 Minimal models

Given an interpretation I, a rule r is *satisfied w.r.t.* I if some head atom is true w.r.t. I whenever all body literals are true w.r.t. I. An interpretation M is a *model* of a DLP$^{\mathscr{A}}$ program $\mathscr{P}$ if all $r \in Ground(\mathscr{P})$ are satisfied w.r.t. M. A model M for $\mathscr{P}$ is (subset) minimal if no model N for $\mathscr{P}$ exists such that $N \subset M$.

2.2.5 Answer sets

We now recall the generalization of the Gelfond-Lifschitz transformation to programs with aggregates from Faber *et al.* (2004).

Definition 2.3 (*Faber et al. 2004*)
Given a ground DLP$^{\mathscr{A}}$ *program* $\mathscr{P}$ *and a total interpretation* I*, let* $\mathscr{P}^I$ *denote the transformed program obtained from* $\mathscr{P}$ *by deleting all rules in which a body literal is false w.r.t.* I*.* I *is an answer set of a program* $\mathscr{P}$ *if it is a minimal model of* $Ground(\mathscr{P})^I$*.*

Example 2.4
Consider the following two programs:

$$P_1 : \{p(a) \text{ :- } \#\texttt{count}\{X : p(X)\} > 0.\} \qquad P_2 : \{p(a) \text{ :- } \#\texttt{count}\{X : p(X)\} < 1.\}$$

$Ground(P_1) = \{p(a) \text{ :- } \#\texttt{count}\{\langle a : p(a)\rangle\} > 0.\}$ and $Ground(P_2) = \{p(a) \text{ :- } \#\texttt{count}\{\langle a : p(a)\rangle\} < 1.\}$; consider also interpretations $I_1 = \{p(a)\}$ and $I_2 = \emptyset$. Then, $Ground(P_1)^{I_1} = Ground(P_1)$, $Ground(P_1)^{I_2} = \emptyset$, and $Ground(P_2)^{I_1} = \emptyset$, $Ground(P_2)^{I_2} = Ground(P_2)$ hold. I_2 is the only answer set of P_1 (because I_1 is not a minimal model of $Ground(P_1)^{I_1}$), whereas P_2 admits no answer set (I_1 is not a minimal model of $Ground(P_2)^{I_1}$, and I_2 is not a model of $Ground(P_2) = Ground(P_2)^{I_2}$). □

Note that any answer set A of $\mathscr{P}$ is also a model of $\mathscr{P}$ because $Ground(\mathscr{P})^A \subseteq Ground(\mathscr{P})$, and rules in $Ground(\mathscr{P}) - Ground(\mathscr{P})^A$ are satisfied w.r.t. A.

3 System functionalities

As pointed out in the Introduction, the presented system allows for two typologies of execution: (i) direct database execution (DLV^{DB}), which is capable of handling

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

```
Auxiliary-Directives ::= Init-section [Table-definition]+ [Query-Section]? [Final-section]*
Init-Section ::=USEDB DatabaseName:UserName:Password [System-Like]?.
Table-definition ::=
[USE TableName [( AttrName [, AttrName]* )]? [AS ( SQL-Statement )]?
[FROM DatabaseName:UserName:Password]?
[MAPTO PredName [( SqlType [, SqlType]* )]? ]?.
|
CREATE TableName [( AttrName [, AttrName]* )]?
[MAPTO PredName [( SqlType [, SqlType]* )]? ]?
[KEEP_AFTER_EXECUTION]?.]
Query-Section ::= QUERY TableName.
Final-section ::=
[DBOUTPUT DatabaseName:UserName:Password.
|
OUTPUT [APPEND | OVERWRITE]? PredName [AS AliasName]?
[IN DatabaseName:UserName:Password.]
System-Like ::= LIKE [POSTGRES | ORACLE | DB2 | SQLSERVER | MYSQL]
```

Fig. 1. Grammar of the auxiliary directives.

massive amounts of data but with some limitations on the expressiveness of the query program (see Section 2), and (ii) main-memory execution (DLV^{IO}), which allows the user to take full advantage of the expressiveness of $\text{DLP}^{\not\vee}$ and to import data residing on DBMSs but with some limitations on the quantity of data to reason about, given by the amount of available main-memory.

The system, along with a manual and some examples, is available for download at `http://www.mat.unical.it/terracina/dlvdb`. In the following, we provide a general description of the main functionalities provided by DLV^{DB} and DLV^{IO}. The interested reader can find all details on the system's Web site.

3.1 Direct database execution

Three main peculiarities characterize the DLV^{DB} system in this execution modality: (i) its ability to evaluate logic programs directly and completely on databases with a very limited usage of main-memory resources; (ii) its capability to map program predicates to (possibly complex and distributed) database views; and (iii) the possibility to easily specify which data are to be considered as input or as output for the program. This is the main contribution of our work.

Roughly speaking, in this execution modality the user has his data stored in (possibly distributed) database tables and wants to carry out some reasoning on them; however, the amount of such data, or the number of facts that are generated during the reasoning, is such that the evaluation cannot be carried out in main-memory. Then, the program must be evaluated directly in mass-memory.

To properly carry out the evaluation, it is necessary to specify the mappings between input and output data and program predicates, as well as to specify whether the temporary relations possibly needed for the mass-memory evaluation should be maintained or deleted at the end of the execution. These can be specified by some auxiliary directives. The grammar in which these directives must be expressed is shown in Figure 1.

Intuitively, the user must specify the working database in which the system has to perform the evaluation. Moreover, he can specify a set of table definitions; note that each specified table is mapped into one of the program predicates. Facts can reside

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

```
USEDB dlvdb:myname:mypasswd.
USE flight_rel (Id, FromX, ToY, Company) FROM dbAirports:airportUser:airportPasswd
MAPTO flight (integer, varchar(255), varchar(255), varchar(255)).
USE codeshare_rel (Company1, Company2, FlightId) FROM dbCommercial:commUser:commPasswd
MAPTO codeshare (varchar(255), varchar(255), integer).
CREATE destinations_rel (FromX, ToY, Company)
MAPTO destinations (varchar(255), varchar(255), varchar(255)) KEEP_AFTER_EXECUTION.
OUTPUT destinations AS composedCompanyRoutes IN dbTravelAgency:agencyName:agencyPasswd.
```

Fig. 2. Auxiliary directives for Example 3.2.

on separate databases or they can be obtained as views on different tables. Attribute type declaration is needed only for a correct management of built-in predicates. The `USE` and `CREATE` options can be exploited to specify input and output data as well as temporary relations needed for the mass-memory instantiation. Finally, the user can choose to copy the entire output of the evaluation or parts thereof in different databases.

Example 3.1

Assume that a travel agency asks to derive all the destinations reachable by an airline company either by using its aircrafts or by exploiting code-share agreements. Suppose that the direct flight plans of each company are stored in a relation `flight_rel(Id, FromX, ToY, Company)` of the database `dbAirports`, whereas the code-share agreements between companies are stored in a relation `codeshare_rel (Company1, Company2, FlightId)` of an external database `dbCommercial`. If a code-share agreement holds between the company $c1$ and the company $c2$ for $flightId$, it means that the flight $flightId$ is actually provided by an aircraft of $c1$ but can be considered also carried out by $c2$. Finally, assume that, for security reasons, travel agencies are not allowed to directly access the databases `dbAirports` and `dbCommercial`, and, consequently, it is necessary to store the output result in a relation `composedCompanyRoutes` of a separate database `dbTravelAgency` supposed to support travel agencies. The DLP$^{\mathcal{A}}$ program that can derive all the connections is:

(1) $destinations(FromX, ToY, Comp)$:- $flight(Id, FromX, ToY, Comp)$.
(2) $destinations(FromX, ToY, Comp)$:- $flight(Id, FromX, ToY, C2)$, $codeshare(C2, Comp, Id)$.
(3) $destinations(FromX, ToY, Comp)$:- $destinations(FromX, T2, Comp)$, $destinations(T2, ToY, Comp)$.

To exploit data residing in the abovementioned databases, we should map the predicate $flight$ to the relation `flight_rel` of `dbAirports` and the predicate *codeshare* to the relation `codeshare_rel` of `dbCommercial`. Finally, we have to map the predicate *destinations* to the relation `composedCompanyRoutes` of `dbTravelAgency`.

Now suppose that, due to a huge size of input data, we need to evaluate the program in mass-memory (on a DBMS). To carry out this task, the auxiliary directives shown in Figure 2 should be used. They allow to specify the mappings between the program predicates and the database relations introduced previously. □

It is worth pointing out that if a predicate is not explicitly mapped into a table, but a relation with the same name and compatible attributes is present in the working database, the system automatically hypothesize a `USE` mapping for them. Analogously, if a predicate is not explicitly mapped and no corresponding table exists

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

in the working database, a `CREATE` mapping is automatically hypothesized for it. This significantly simplifies the specification of the auxiliary directives; in fact, in the ideal case—when everything is in the working database and each input predicate has the corresponding input table with the same name—only the `Init-Section` and one of `CREATE` or `OUTPUT` options are actually needed to run a program and check its output.

3.2 Main-memory execution

The main-memory execution modality of the system allows input facts to be (possibly complex) views on database tables and allows exporting (parts of) predicates to database relations. However, the program evaluation is carried out completely in main-memory; this allows the system to evaluate more complex logic programs (see Section 2) but at the price of a lower amount of data the system can handle, due to the limited amount of main-memory.

The concept of importing and exporting data from external data sources into logic-based systems is not new (see, for example, Lu *et al.* 1996; Rao *et al.* 1997; Arni *et al.* 2003); the contribution of this execution modality is mainly of technological relevance and has the merit of providing Answer Set Programming with an easy way to access distributed data spread over the network. Another advancement with respect to existing proposals is its flexibility in the types of external source that can be accessed; in fact, most of the existing systems are tailored on custom DBMSs, whereas our system can be interfaced with any external source that provides an ODBC connection.

Intuitively, DLV^{IO} can be exploited when the user has to perform very complex reasoning tasks (in the NP class or higher) but the data are available in database relations, or the output must be permanently stored in a database for further elaborations.

To perform these tasks, two built-in commands are added in DLV^{IO} to the standard DLP$^{\mathscr{A}}$ syntax, namely, the #import and the #export commands:

```
#import(databasename,"username","password","query",predname, typeConv).
#export(databasename,"username","password",predname,tablename).
```

An #import command retrieves data from a table "row by row" through the *query* specified by the user in SQL and creates one atom for each selected tuple. The name of each imported atom is set to *predname*, and is considered as a fact of the program.

The #export command generates a new tuple into *tablename* for each new truth value derived for *predname* by the program evaluation.

An alternative form of the #export command is the following:

```
#export(databasename, "username", "password", predname, tablename,
                                        "REPLACE where <condition>" )
```

which can be used to remove from *tablename* the tuples of *predname* for which the "REPLACE where" condition holds; it can be useful for deleting tuples corresponding to violated integrity constraints.

It is worth pointing out that if a DLP$^{\mathscr{A}}$ program contains at least one #export command, the system can compute only the first valid answer set; this limitation has been introduced mainly to avoid an exponential space complexity of the system. In fact, the number of answer sets can be exponential in the input.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

Example 3.2
Consider again the scenario introduced in Example 3.1, and assume that the amount of input data allows the evaluation to be carried out in main-memory. The built-in commands that must be added to the DLP$^{\mathcal{A}}$ program of Example 3.1 to implement the necessary mappings are:

```
#import(dbAirports, "airportUser", "airportPasswd" , "SELECT * FROM flight_rel",
                                    flight, type : U_INT, Q_CONST, Q_CONST, Q_CONST).
#import(dbCommercial, "commUser", "commPasswd", "SELECT * FROM codeshare_rel",
                                    codeshare, type : Q_CONST, Q_CONST, U_INT).
#export(dbTravelAgency, "agencyName", "agencyPasswd", destinations,
                                    composedCompanyRoutes).
```
□

Note that the syntax of DLV^{IO} directives is simpler than that of DLV^{DB} auxiliary directives. This is because DLV^{IO} is intended to provide an easy mechanism to load data into the logic program and then store its results back to mass-memory, whereas DLV^{DB} is oriented to more sophisticated applications handling distributed data and mass-memory-based reasoning, and, consequently, it must provide a richer set of options in defining the mappings.

4 Implementation principles

The main innovation of our system resides in the evaluation of DLP$^{\mathcal{A}}$ programs directly on a database. The evaluation process basically consists of two steps: (i) the translation of DLP$^{\mathcal{A}}$ rules in SQL statements, (ii) the definition of an efficient SQL query plan such that the computed answers are the same as the ones of the main-memory execution, but where the evaluation process is completely carried out in mass-memory. In the following, we first describe the general philosophy of our mass-memory evaluation strategy, then we present the algorithms used to obtain SQL statements from DLP$^{\mathcal{A}}$ rules.

4.1 General characteristics of the evaluation strategy

The evaluation of a program $\mathscr{P}$ starts from the analysis of its intensional component. In particular, $\mathscr{P}$ is first transformed into an equivalent program $\mathscr{P}'$ that can be evaluated more efficiently by the subsequent steps. Transformations carried out in this phase take into account various aspects of the input program; as an example, they aim to (i) reduce the arity of intermediate relations whenever possible, (ii) reduce the size of intermediate relations (Faber *et al.* 1999a), (iii) push down constants in the queries by magic sets rewritings (Bancilhon *et al.* 1986; Ross 1990; Beeri and Ramakrisnhan 1991; Mumick *et al.* 1996). All these optimizations do not take into account the extensional component (the facts) of $\mathscr{P}$; some other optimizations are described in Faber *et al.* (1999a).

After this, the connected components and their topological order (i.e., the Dependency Graph (DG)) of the resulting program are computed. Then, it is evaluated one component at a time, starting from the lowest ones in the topological order.

The evaluation of each component follows the Semi-Naive method (Ullman 1989) with the enhancements showed in Balbin and Ramamohanarao (1987) and Zaniolo *et al.* (1997) to optimize the evaluation of rules with non-linear recursion.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

In particular, the Semi-Naive algorithm applied to a component $\mathscr{P}_C$ can be viewed as a two-phase algorithm: the first one deals with non-recursive rules, which can be completely evaluated in one single step; the second one deals with recursive rules, which need an iterative fixpoint computation for their complete evaluation. At each iteration, there are a number of predicates whose extensions have been already fully determined (predicates not belonging to $\mathscr{P}_C$ that have been therefore previously evaluated), and a number of recursive predicates (i.e., belonging to $\mathscr{P}_C$) for which a new set of truth values can be determined from the available ones. Then, in order to evaluate, for example, the rule:

$$(r_1): \quad p_0(X,Y) \text{ :- } p_1(X,Y), p_2(Y,Z), q(X,Z),$$

where p_1 and p_2 are mutually recursive with p_0 and q is not recursive, the standard Semi-Naive method evaluates the following formula (expressed in relational algebra) at each iteration:

$$\begin{array}{lccccccc} \Delta P_0^k = & \Delta P_1^{k-1} & \bowtie & P_2^{k-1} & \bowtie & Q & \cup & (a) \\ & P_1^{k-1} & \bowtie & \Delta P_2^{k-1} & \bowtie & Q & & (b) \end{array}$$

Here, a capital letter is used to indicate the database relation corresponding to the (lower case) predicate; P_j^k indicates the values stored in relation P_i up to step k and ΔP_j^k is the set of new values determined for P_j at step k (in the following, we call ΔP_j^k the *differential* of P_j).

However, the standard Semi-Naive approach is characterized by inefficiencies in evaluating non-linear recursive rules. In fact, if each P_j^{k-1} is expanded in its (disjoint) components P_j^{k-2} and ΔP_j^{k-1}, the formula $(a) \cup (b)$ above becomes:

$$\Delta P_0^k = \begin{array}{ll} \text{(a)} \left[\begin{array}{lccccccc} & \Delta P_1^{k-1} & \bowtie & P_2^{k-2} & \bowtie & Q & \cup & (1) \\ & \Delta P_1^{k-1} & \bowtie & \Delta P_2^{k-1} & \bowtie & Q & \cup & (2) \end{array} \right. \\ \text{(b)} \left[\begin{array}{lccccccc} & P_1^{k-2} & \bowtie & \Delta P_2^{k-1} & \bowtie & Q & \cup & (3) \\ & \Delta P_1^{k-1} & \bowtie & \Delta P_2^{k-1} & \bowtie & Q & & (4) \end{array} \right. \end{array}$$

where (a) expands P_2^{k-1} and (b) expands P_1^{k-1}; note that line (2) and (4) are identical. The enhancement described in Balbin and Ramamohanarao (1987) and Zaniolo *et al.* (1997) provides a solution to this problem rewriting the original rule in:

$$\begin{array}{lccccccc} \Delta P_0^k = & \Delta P_1^{k-1} & \bowtie & P_2^{k-1} & \bowtie & Q & \cup \\ & P_1^{k-2} & \bowtie & \Delta P_2^{k-1} & \bowtie & Q & \end{array}$$

which, indeed, avoids to re-compute joins in (2), (4) more times.

Generalizing the solution to a rule having r predicates mutually recursive with its head, the differentiation is obtained by subdividing the original rule in r subrules such that the i-th subrule has the form $\Delta p_0^k : -p_1^{k-2}, \ldots, p_{i-1}^{k-2}, \Delta p_i^{k-1}, p_{i+1}^{k-1}, \ldots, p_r^{k-1}, q$. Note that both relations P_j^{k-1}, P_j^{k-2}, and ΔP_j^{k-1} are considered.

In our approach, we follow a slightly different strategy, which both unfolds *each* relation P_j^{k-1} in P_j^{k-2} and ΔP_j^{k-1} and avoids to produce the redundant subrules of the standard Semi-Naive method. This is carried out as follows. Let us tag the differential relations (ΔP_j^{k-1}) with the symbol 1 and the standard ones (P_j^{k-2}) with the symbol 0. Given a generic rule with r predicates $p_1, \ldots, p_r$ in its body mutually recursive with the head, our approach follows the binary enumeration between 1

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

Differential Semi-Naive(Input: $R_1,\ldots,R_l$**. Output:** $Q_1,\ldots,Q_m$**,** $P_1,\ldots,P_n$**)**
begin
  for i:=1 **to** m **do** // Evaluate non recursive predicates
(1) $Q_i = EVAL(q_i,\ R_1,\ldots,\ R_l,\ Q_1,\ldots,\ Q_m)$;
  for i:=1 **to** n **do begin** // Initialize recursive predicates
(2) $P_i^{k-2} = EVAL(p_i,\ R_1,\ldots,\ R_l,\ Q_1,\ldots,\ Q_m)$;
(3) $\Delta P_i^{k-1} = P_i^{k-2}$;
  end;
  repeat
    for i:=1 **to** n **do begin**
(4) $\Delta P_i^k = EVAL\_DIFF(p_i, P_1^{k-2},\ldots,P_n^{k-2},\Delta P_1^{k-1},\ldots,\Delta P_n^{k-1},R_1,\ldots,\ R_l,Q_1,\ldots,\ Q_m)$;
(5) $\Delta P_i^k = \Delta P_i^k - P_i^{k-2} - \Delta P_i^{k-1}$;
    end;
    for i:=1 **to** n **do begin**
(6) $P_i^{k-2} = P_i^{k-2} \cup \Delta P_i^{k-1}$;
(7) $\Delta P_i^{k-1} = \Delta P_i^k$;
    end;
  until $\Delta P_i^k = \emptyset, \forall i\ 1 \leqslant i \leqslant n$;
  for i:=1 **to** n **do**
(8) $P_i = P_i^{k-2}$;
end.

Fig. 3. Algorithm differential Semi-Naive.

and $2^r - 1$, and, for each of these binary numbers, it generates a differential rule; in particular, if position j on the binary number contains a 0, then P_j^{k-2} is put in the corresponding rule, otherwise ΔP_j^{k-1} is used. As for the previous example, rule (r_1) is evaluated, in our approach, with joins (1), (2), and (3) shown above.

Note that this approach generates a higher number of auxiliary rules with respect to Balbin and Ramamohanarao (1987) and Zaniolo *et al.* (1997), but, while avoiding to execute the same set of redundant joins, it allows handling smaller relations. This could constitute a good advantage when handling massive amounts of data, because managing several small joins can be less resource demanding in comparison with executing few big ones.

The algorithm implemented in our system for the differential Semi-Naive evaluation strategy described above is shown in Figure 3. It is executed for each component $\mathscr{P}_C$ of the input program $\mathscr{P}$ and assumes that input DLP$^{\mathscr{A}}$ rules have been already translated to SQL statements. Here, the component $\mathscr{P}_C$ depends on predicates $r_1,\ldots,r_n$ solved in previous components and has $q_1,\ldots,q_m$ as non-recursive predicates or facts and $p_1,\ldots,p_n$ as recursive predicates.

Function $EVAL(q_i,\ R_1,\ldots,\ R_l,\ Q_1,\ldots,\ Q_m)$ performs the evaluation of the non-recursive rules having q_i as head as follows: it first runs each SQL query corresponding to a rule having q_i as head; then, the corresponding results are added to the relation Q_i.

Function $EVAL\_DIFF(p_i, P_1^{k-2},\ldots,P_n^{k-2},\Delta P_1^{k-1},\ldots,\Delta P_n^{k-1},R_1,\ldots,R_l,Q_1,\ldots,Q_m)$ implements the optimization to the Semi-Naive method; it computes the new values for the predicate p_i at the current iteration k starting from the values computed until iteration $k-2$ and the new values obtained at the previous iteration $k-1$. In more detail, the SQL statements corresponding to each recursive rule having p_i as head are considered. The final result of $EVAL\_DIFF$ is stored in table ΔP_i^k. Clearly, it

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

cannot be proved that *EVAL_DIFF* does not recompute some truth values already obtained in previous iterations. As a consequence, ΔP_i^k must be cleaned up from these values after the computation of *EVAL_DIFF*; this is exactly what is done by instruction (5) of the algorithm. Instructions (6) and (7) are needed to reuse the same relations (ΔP_i^k, ΔP_i^{k-1}, P_i^{k-2}) at each iteration.

Finally, it is worth pointing out that the last **for** of the algorithm (instruction (8)) is shown just for clarity of exposition; in fact, in the actual implementation, what we indicated as P_i^{k-2} is exactly table P_i.

It is worth pointing out that the basic step of the evaluation is the execution of standard SQL queries over the underlying data. In fact, one of the main objectives in the implementation of DLV^{DB} has been that of associating one single (non-recursive) SQL statement with each rule of the program (either recursive or not), without the support of main-memory data structures for the evaluation. This allows DLV^{DB} to minimize the "out of memory" problems caused by limited main-memory dimensions. Moreover, the overall organization of the evaluation strategy allows benefiting from both the optimizations on the intensional component of the program (the program rewriting techniques outlined at the beginning of this section) and the optimizations on the extensional component (the data) already implemented in the DBMS configured as the working database.

The combination of such optimizations, along with a wise translation of datalog rules in efficient SQL queries allow DLV^{DB} to boost the evaluation process even with respect to main-memory evaluation strategies (see Section 6).

4.2 *From* DLP$^{\mathscr{A}}$ *to SQL*

In this section, we describe the general functions exploited to translate DLP$^{\mathscr{A}}$ rules in SQL statements. Functions are presented in pseudocode and, for the sake of presentation clarity, they omit some details. Moreover, since there is a one-to-one correspondence between the predicates in the logic program and the relations in the database, in the following, when this is not confusing, we use the terms predicate and relation interchangeably. It is worth recalling that these one-to-one correspondences are determined both from the user specifications in the auxiliary directives and from the mappings automatically derived by the system.

To provide examples for the presented functions, we exploit the following reference schema:

$$employee(Ename, Salary, Dep, Boss) \quad department(Code, Director)$$

storing information about the employees of the departments of a given company. Specifically, each employee has associated a *Boss* who is, in his turn, an employee.

4.2.1 *Translating non-recursive rules*

Non-recursive rules are translated in a quite standard way in SQL. The only exceptions are made for rules containing aggregate functions and rules containing built-ins. The general format of the SQL statement generated in the translation is:

INSERT INTO head(r) (Translate_SQL(r))

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

where *head(r)* returns the relation associated with the head of r; this task is carried out by considering the mappings specified in the auxiliary directives. *Translate_SQL(r)* takes into account the kind of rule (e.g., if it contains negation or built-ins) and calls the suitable transformation function. These functions are described next.

4.2.2 Translating positive rules

Intuitively, the SQL statement for positive rules is composed as follows: the SELECT part is determined by the variable bindings between the head and the body of the rule. The FROM part of the statement is determined by the predicates composing the body of the rule; variable bindings between body atoms and constants determine the WHERE conditions of the statement. Finally, an EXCEPT part is added in order to eliminate tuple duplications. The behavior of function *TranslatePositiveRule* is well described by the following example:

Example 4.1
Consider the following rule:

$$q_0(Ename) \text{ :- } employee(Ename, 100.000, Dep, Boss), department(Dep, rossi).$$

which returns all the employees working at the department whose chief is *rossi* and having a yearly salary of *100.000* euros. The corresponding SQL statement is the following:[5]

```
INSERT INTO q0 (
 SELECT  employee.att1 FROM employee,  department
 WHERE employee.att3  =  department.att1 AND department.att2='rossi'
    AND employee.att2=100.000 EXCEPT (SELECT * FROM q0))
```

□

4.2.3 Translating rules with negated atoms

Intuitively, the construction of the SQL statement for this kind of rule is carried out as follows: the positive part of the rule is handled in a way very similar to what has been shown for function *TranslatePositiveRule*; then, each negated atom is handled by a corresponding NOT IN part in the statement. The behavior of function *TranslateRuleWithNegation* is well illustrated by the following example:

Example 4.2
The following program computes (using the goal *topEmployee*) the employees that have no other boss than the director.

$$\begin{aligned} topEmployee(Ename) \quad &:- \quad employee(Ename, Salary, Dep, Boss), \\ &\qquad\quad department(Dep, Boss), \\ &\qquad\quad \text{not } otherBoss(Ename, Boss). \\ otherBoss(Ename, Boss) \quad &:- \quad employee(Ename, Salary, Dep, Boss), \\ &\qquad\quad employee(Boss, Salary, Dep, Boss1). \end{aligned}$$

[5] Here and in the following we use the notation $t.att_i$ to indicate the i-th attribute of the table t. Actual attribute names are determined from the auxiliary directives.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

The first rule above is translated into the following SQL statement:

```
INSERT INTO topEmployee (
 SELECT employee.att1 FROM employee, department
 WHERE (employee.att3=department.att1) AND (employee.att4=department.att2)
    AND (employee.att1, employee.att4)
    NOT IN (SELECT otherBoss.att1, otherBoss.att2 FROM otherBoss )
 EXCEPT (SELECT * FROM topEmployee))
```

□

4.2.4 Translating rules with built-in predicates

As pointed out in Section 2, in addition to user-defined predicates, some comparative and arithmetic predicates are provided by the reasoning language. When running a program containing built-in predicates, the range of admissible values for the corresponding variables must be fixed. We map this necessity in the working database by adding a restriction based on the maximum value allowed for variables. Moreover, in order to allow mathematical operations among attributes, DLV^{DB} requires the types of attributes to be properly defined in the auxiliary directives.

The function for translating rules containing built-in predicates is a slight variation of the function for translating positive rules. As a matter of fact, the presence of a built-in predicate in the rule implies just adding a corresponding condition in the WHERE part of the statement. However, if the variables specified in the built-in are not bound to any other variable of the atoms in the body, a #maxint value must be exploited to bound that variable to its admissible range of values.

Example 4.3
The program:

$$q_1(Ename) : -employee(Ename, Salary, Dep, Boss),\ Salary > 100.000.$$

is translated to the SQL statement:

```
INSERT INTO q1
 (SELECT employee.att1 FROM employee WHERE employee.att2 > 100.000
  EXCEPT (SELECT * FROM q1))
```

□

4.2.5 Translating rules with aggregate atoms

In Section 2, we introduced the syntax and the semantics of DLP with aggregates. We have also shown that specific safety conditions must hold for each rule containing aggregate atoms in order to guarantee the computability of the corresponding rule. As an example, aggregate atoms cannot contain predicates mutually recursive with the head of the rule they are placed in; from our point of view, this implies that the truth values of each aggregate function can be computed once and for all before evaluating the corresponding rule (which can be, in its turn, recursive).

Actually, the process that rewrites input programs before their execution, automatically rewrites each rule containing some aggregate atom in such a way that it follows a standard format (we call this process standardization in the following).

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

```
Function TranslateAggregateRule(VAR r: DLP^¬ rule): SQL statement
begin
    for each a in aggr_atom(r) do begin
        aux:=aux_atom(a);
        SQL:="CREATE VIEW " + aux +"_supp" +
            "AS (SELECT "+ bound_attr(a) + ", " +
                aggr_func(a) + "(" + aggr_attr(a) + ") " +
            "FROM " + aux + "GROUP BY " + bound_attr(a) + ")";
        removeFromBody(r, a);
        addToBody(r, aux_atom_supp(a));
        addToBody(r, guards(a));
    end;
    return SQL;
end.
```

Fig. 4. Function *TranslateAggregateRule*.

Specifically, given a generic rule of the form:

$$head \quad :- \quad body, f(\{Vars : Conj\}) \prec Rg.$$

where $Conj$ is a generic conjunction and Rg is a guard, the system automatically translates this rule to a pair of rules of the form

$$\begin{array}{lll} auxAtom & :- & Conj, BindingAtoms. \\ head & :- & body, f(\{Vars : auxAtom\}) \prec Rg. \end{array}$$

where *auxAtom* is a standard rule containing both $Conj$ and the atoms (*BindingAtoms*) necessary for the bindings of $Conj$ with *body* and/or *head*. Note that *auxAtom* contains only those attributes of $Conj$ that are strictly necessary for the computation of f and, consequently, it may have far less (and cannot have more) attributes in comparison with those present in $Conj$.

In our approach, we rely on this standardization to translate this kind of rule to SQL; clearly only the second rule, containing the aggregate function, is handled by the function we are presenting next; in fact, the first rule is automatically translated by one of the already presented functions.

Intuitively, the objective of our translation is to create an SQL view *auxAtom_supp* from *auxAtom*, which contains all the attributes necessary to bind *auxAtom* with the other atoms of the original rule and a column storing the results of the computation of f over *auxAtom*; the original aggregate atom is then replaced by this view and guard conditions are suitably translated by logic conditions between variables. At this point, the resulting rule is a standard rule not containing aggregate functions and can be then translated by one of the functions we have presented previously; clearly enough, in this process, the original input rule r must be modified to have a proper translation of its "standard" part. The function is shown in Figure 4; it receives a rule r with aggregates as input and returns both the SQL views for the aggregate functions in r and the modified (standard) r, which will be handled by standard translation functions.[6]

[6] Here and in the following, we use the operator + to denote the "append" operator between strings.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

Here function *aggr_atom(r)* returns the aggregate atoms present in r; *aux_atom(a)* returns the auxiliary atom corresponding to $Conj$ of a and automatically generated by the standardization. Function *bound_attr(a)* yields in output the attributes of the atom a bound with attributes of the other atoms in the rule, whereas *aggr_attr(a)* returns the attribute that the aggregation must be carried out on to (the first variable in $Vars$). *aggr_func(a)* returns the SQL aggregation statement corresponding to the aggregate function of a. Functions *removeFromBody* and *addToBody* are responsible of altering the original rule r to make it standard (without aggregates). In particular, *removeFromBody(r, a)* removes the aggregate atom a from the rule r, whereas *addToBody* adds both *aux_atom_supp(a)* and *guards(a)* to r. Note that *aux_atom_supp(a)* yields in output the name of the atom corresponding to the just created auxiliary view, whereas *guards(a)* converts the guard of the aggregate atom a in a logic statement between attributes in the rule.

Example 4.4

Consider the following rule computing the departments which spend for the salaries of their employees, an amount greater than a certain threshold, say 100,000:

$$costlyDep(Dep) \text{ :- } department(Dep, \_), \#sum\{Salary, Ename : employee(Ename, Salary, Dep, \_)\} > 100000.^{7}$$

The standardization automatically rewrites this rule as:

$$aux\_emp(Salary, Ename, Dep) \text{ :- } department(Dep, \_), employee(Ename, Salary, Dep, \_).$$
$$costlyDep(Dep) \text{ :- } department(Dep, \_), \#sum\{Salary, Ename : aux\_emp(Salary, Ename, Dep)\} > 100000.$$

The first rule is treated as a standard positive rule and is translated to:

```
INSERT INTO aux_emp (
 SELECT employee.att2, employee.att1, department.att1
 FROM department, employee WHERE department.att1 = employee.att3
 EXCEPT (SELECT * FROM aux_emp))
```

The second rule is translated to:

```
CREATE VIEW aux_emp_supp AS (
 SELECT aux_emp.att3, SUM (aux_emp.att1) FROM aux_emp
 GROUP BY aux_emp.att3)

INSERT INTO costlyDep (
 SELECT department.att1 FROM department, aux_emp_supp
 WHERE department.att1 = aux_emp_supp.att1 AND aux_emp_supp.att2 > 100000
 EXCEPT (SELECT * FROM costlyDep))
```

□

[7] Note that Ename is needed to sum also the salaries of employees earning the same amount (see the discussion on sets/multisets in Dell'Armi *et al.* 2003a).

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

```
Function TranslateRecursiveRule(r: DLP^A rule ): SQL statement
begin
    SQL:="";
    if(hasAggregate(r))
        SQL:=TranslateAggregateRule(r);
    n:=2^RecursivePredicates(r)-1
    SQL:=SQL+"INSERT INTO " + Δhead(r) + "(";
    for i:=1 to n do begin
        Let r' be a rule;
        setHead(r', Δhead(r));
        for each non recursive predicate q_j in body(r) do
            addToBody(r', q_j);
        for each recursive predicate p_j in body(r) do
            if (bit(j,i)=0) then addToBody(r', p_j^{k-2});
            else addToBody(r', Δp_j^{k-1});
        if (i ≠ 1)  SQL:=SQL+"UNION ";
        SQL:=SQL + TranslateNonRecursiveRule(r');
    end;
    SQL:=SQL + ")";
    return SQL;
end.
```

Fig. 5. Function *TranslateRecursiveRule*.

4.2.6 Translating recursive rules

As previously pointed out, our program evaluation strategy exploits a refined version of the Semi-Naive method. This is based on the translation of a recursive rule into a non-recursive SQL statement operating alternatively on standard and differential versions of the relations associated with recursive predicates. Each time this statement is executed by the algorithm, it must compute just the new values for the predicate in the head that can be obtained from the values computed in the last two iterations of the fixpoint.

Intuitively, the translation algorithm must first select the proper combinations of standard and differential relations from the rule r under consideration; then, for each of these combinations, it must rewrite r in a corresponding rule r'. Each r' thus obtained is nonrecursive, and, consequently, it can be handled by Function *TranslateNonRecursiveRule*. Algorithm *TranslateRecursiveRule* is shown in Figure 5.

Here, functions *TranslateAggregateRule* and *TranslateNonRecursiveRule* have been introduced previously. Function *hasAggregate(r)* returns true if r contains aggregate functions. Function *RecursivePredicates(r)* returns the number of occurrences of recursive predicates in the body of r; *Δhead(r)* returns the differential version of the relation corresponding to the head of r. Function *setHead(r′, p)* sets the head of the rule r' to the predicate p; analogously, function *addToBody(r′, p)* adds to the body of r' a conjunction with the predicate p. Function *bit(j,i)* returns the j-th bit of the binary representation of i.

It is worth noticing that the execution of the queries resulting from function *TranslateRecursiveRule* implement function *EVAL_DIFF* for r (see the algorithm of Figure 3).

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

Example 4.5

Consider the situation in which we need to know whether the employee e_1 is the boss of the employee e_n either directly or by means of a number of employees $e_2, .., e_n$ such that e_1 is the boss of e_2, e_2 is the boss of e_3, etc. Then, we have to evaluate the program:

$$\begin{array}{lllll} r_1: & q_2(E_1, E_2) & :- & employee(E_1, Salary, Dep, E_2). \\ r_2: & q_2(E_1, E_3) & :- & q_2(E_1, E_2),\ q_2(E_2, E_3). \end{array}$$

containing the recursive rule r_2. This program cannot be evaluated in one single iteration of the Semi-Naive computation. Rule r_1 is not recursive; it is translated by Function *TranslatePositiveRule* to the following SQL that is evaluated once:

```
INSERT INTO q2 ( SELECT employee.att1, employee.att4 FROM employee
 EXCEPT (SELECT * FROM q2))
```

Rule r_2 is first translated by Function *TranslateRecursiveRule* to the temporary set of rules:

$$\begin{array}{lllll} r_2': & \Delta q_2^k(E_1, E_3) & :- & q_2^{k-2}(E_1, E_2),\ \Delta q_2^{k-1}(E_2, E_3). \\ & \Delta q_2^k(E_1, E_3) & :- & \Delta q_2^{k-1}(E_1, E_2),\ q_2^{k-2}(E_2, E_3). \\ & \Delta q_2^k(E_1, E_3) & :- & \Delta q_2^{k-1}(E_1, E_2),\ \Delta q_2^{k-1}(E_2, E_3). \end{array}$$

which is translated to:

```
INSERT INTO Δq2^k (
 SELECT q2^{k-2}.att1, Δq2^{k-1}.att2 FROM q2^{k-2},Δq2^{k-1} WHERE (q2^{k-2}.att2=Δq2^{k-1}.att1)
 EXCEPT (SELECT * FROM Δq2^k)
 UNION
 SELECT Δq2^{k-1}.att1, q2^{k-2}.att2 FROM Δq2^{k-1}, q2^{k-2} WHERE (Δq2^{k-1}.att2=q2^{k-2}.att1)
 EXCEPT (SELECT * FROM Δq2^k)
 UNION
 SELECT Δq2^{k-1}.att1, Δq2^{k-1}_1.att2 FROM Δq2^{k-1}, Δq2^{k-1} AS Δq2^{k-1}_1
 WHERE (Δq2^{k-1}.att2=Δq2^{k-1}_1.att1)
 EXCEPT (SELECT * FROM Δq2^k))
```

Actually, the real implementation of this function also adds, for performance reasons, the following parts to the statement above:

```
EXCEPT (SELECT * FROM Δq2^{k-1})
EXCEPT (SELECT * FROM q2^{k-2})
```

Note that, following *Differential Semi-Naive* algorithm (Figure 3), q_2^{k-2} and Δq_2^{k-1} are first initialized with the result of the evaluation of r_1 (stored in q_2—see instructions (2) and (3) in Figure 3). Then, the SQL above is iteratively executed until the fixpoint is reached. Note that, the aforementioned process executes instructions (1)–(5) of the algorithm in Figure 3. The update of q_2^{k-2} and Δq_2^{k-1} from one iteration to the subsequent one is carried out by instructions (6) and (7) in a straightforward way. □

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

4.2.7 A complete example

Example 4.6

Consider the datalog program presented in Example 3.1 and the mappings shown in Figure 2. The complete query plan derived by DLV^{DB} for them is:

```
(1) INSERT INTO destinations_rel
    (SELECT f.FromX, f.ToY, f.Company FROM flight_rel AS f)

(2) INSERT INTO destinations_rel
    (SELECT f.FromX, f.ToY, c.Company2 FROM flight_rel AS f, codeshare_rel AS c
    WHERE (f.Id=c.FlightId) AND (f.Company=c.Company1)
    EXCEPT (SELECT * FROM destinations_rel))

(3) INSERT INTO d_destinations_rel
    (SELECT d1.FromX, d2.ToY, d1.Company
    FROM d1_destinations_rel AS d1, destinations_rel AS d2
    WHERE (d1.ToY=d2.FromX) AND (d1.Company=d2.Company)
    UNION
    SELECT d1.FromX, d2.ToY, d1.Company
    FROM destinations_rel AS d1, d1_destinations_rel AS d2
    WHERE (d1.ToY=d2.FromX) AND (d1.Company=d2.Company)
    UNION
    SELECT d1.FromX, d2.ToY, d1.Company
    FROM d1_destinations_rel AS d1, d1_destinations_rel AS d2
    WHERE (d1.ToY=d2.FromX)AND (d1.Company=d2.Company)
    EXCEPT (SELECT * FROM d1_destinations_rel)
    EXCEPT (SELECT * FROM destinations_rel)
    EXCEPT (SELECT * FROM d_destinations_rel))
```

SQL statements (1) and (2) are executed only once, since they correspond to non-recursive rules. On the contrary, the statement (3) is executed several times, until the least fixpoint is reached, that is, `d_destinations_rel` is empty. Note that `d_destinations_rel` and `d1_destinations_rel` correspond respectively to $\Delta head(r)$ and Δp^{k-1} introduced in Function *TranslateRecursiveRule*; as shown in Section 4.1 the evaluation algorithm suitably updates the tuples of `destinations_rel` from the new values derived at each iteration in `d_destinations_rel`. □

5 System architecture

In this section, we present the general architecture of our system. It has been designed as an extension of the DLV system (Leone *et al.* 2006), which allows both the evaluation of logic programs directly on databases and the handling of input and output data distributed on several databases. It combines the expressive power of DLV (and the optimization strategies implemented in it) with the efficient data management features of DBMSs (Garcia-Molina *et al.* 2000).

As previously pointed out, the system provides two, quite distinct, functioning modalities, namely, the direct database execution and the main-memory execution modality. In the following, we present the two corresponding architectures separately.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

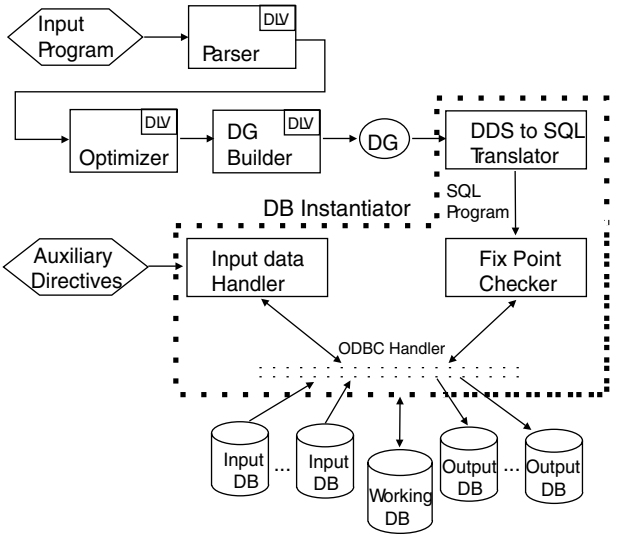

Fig. 6. Architecture of DLV^{DB}.

5.1 Architecture of the direct database execution (DLV^{DB})

Figure 6 illustrates the architecture of the system for the direct database execution. In the figure, the boxes marked with DLV are the ones already developed in the DLV system. An input program $\mathscr{P}$ is first analyzed by the Parser, which encodes the rules in the intensional database (IDB) in a suitable way and builds an extensional database (EDB) in main-memory data structures from the facts specified directly in the program (if any). As for facts already stored in database relations, no EDB is produced in main-memory. After this, the Optimizer applies a rewriting procedure in order to get a program $\mathscr{P}'$, equivalent to $\mathscr{P}$, that can be evaluated more efficiently; some of the operations carried out by this module have been highlighted in Section 4.1. The Dependency Graph Builder computes the DG of $\mathscr{P}'$, its connected components, and a topological ordering of these components. Finally, the DB Instantiator module, the core of the system, is activated.

The DB Instantiator module receives (i) the IDB and the EDB (if not empty) generated by the parser, (ii) the DG generated by the DG builder, and (iii) the auxiliary directives specifying the needed interactions between DLV^{DB} and the databases. It evaluates the input program through the bottom-up fixpoint evaluation strategy shown in Section 4. Since the input program is supposed to be normal and stratified (see Section 2), the DB Instantiator evaluates completely the program and no further modules must be employed after it.

All the instantiation steps are performed directly on the working database through the execution of SQL statements and no data are loaded in main-memory from the databases in any phase of the process. This allows DLV^{DB} to be completely independent of the dimension of both the input data and the number of facts generated during the evaluation.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

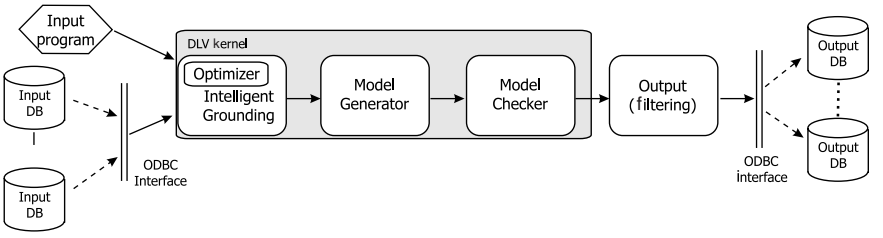

Fig. 7. Architecture of DLV^{IO}.

Communication with databases is performed via ODBC. This allows DLV^{DB} both to be independent from a particular DBMS and to handle databases distributed over the Internet.

It is important to point out that the architecture of DLV^{DB} has been designed in such a way to fully exploit optimizations both from logic theory and from database theory. In fact, the actually evaluated program is the one resulting from the Optimizer module, which applies program rewriting techniques aiming to simplify the evaluation process and to reduce the dimension of the involved relations (see Section 4.1). Then, the execution of the SQL statements in the query plan exploit data-oriented optimizations implemented in the DBMS. As far as this latter point is concerned, we have experienced that the kind of DBMS handling the working database for DLV^{DB} may significantly affect system performance; in fact, when DLV^{DB} was coupled with highly sophisticated DBMSs, it generally showed better performance in handling large amounts of data with respect to the same executions when coupled with less sophisticated DBMSs.

The observation above points out both the importance of data-oriented optimizations and a potential advantage of DLV^{DB} with respect to deductive systems operating on ad hoc DBMSs. In fact, DLV^{DB} can be easily coupled with the most efficient DBMS available at the time being used (provided that it supports standard SQL), whereas the improvement of an ad hoc DBMS is a more difficult task.

5.2 Architecture of the main-memory execution (DLV^{IO})

The architecture of DLV^{IO} is illustrated in Figure 7. It extends the classical DLV architecture with ODBC functionalities to import/export data from/to database relations. The main-memory execution modality acts just as an interface (based on ODBC connections) between the external databases and the standard DLV program.

In more detail, input data can be supplied both by regular files and by relational tables accessed via ODBC as specified by the #import commands. Specifically, for each #import command the system retrieves data from the corresponding table "row by row" through the SQL query specified by the user and creates one atom in main-memory (in the format required by DLV) for each selected tuple. The name of each imported atom is set to *predname*, and is considered as a fact. Possible facts residing in text files are fed into DLV in the standard way. All the data are fetched in main-memory before any evaluation task is carried out.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

The DLV kernel (the shaded part in the figure) then produces answer sets one at a time. It consists of three major components: the "Intelligent Grounding",[8] the "Model Generator", and the "Model Checker" modules; these share a main data structure, the "Ground Program". It is created by the Intelligent Grounding using differential (and other advanced) database techniques together with suitable main-memory data structures, and used by the Model Generator and the Model Checker. The Ground Program is guaranteed to have exactly the same answer sets as the original program. For some syntactically restricted classes of programs (e.g., stratified programs), the Intelligent Grounding module already computes the corresponding answer sets.

For harder problems, most of the computation is performed by the Model Generator and the Model Checker. Roughly, the former produces some "candidate" answer sets (models) (Faber *et al.* 1999b, 2001), the stability and minimality of which are subsequently verified by the latter.

The Model Checker verifies whether the model at hand is an answer set. This task is very difficult in general, because checking the stability of a model is known to be co-NP complete. However, this module exploits the fact that minimal model checking—the hardest part—can be efficiently performed for the relevant class of *head-cycle-free* (HCF) programs (Ben-Eliyahu and Dechter 1994, 1996).

Each time an answer set M is found, "Filtering" is invoked, which performs some post-processing, controls continuation or abortion of the computation, and possibly stores the output data in the corresponding relational tables as specified by the #export commands. In particular, if an #export command from *predname* to *tablename* is present, the module generates a new tuple in *tablename* for each atom in M having name *predname*.[9]

6 Experiments and benchmarks

In this section, we present our experimental framework and the results obtained comparing the DLV^{DB} system with several state-of-the-art systems. Benchmarks have been designed following the guidelines, problems, and data structures proposed in Bancilhon and Ramakrishnan (1988) and Greco (2003) to assess the performance of DDSs. Roughly speaking, problems used in Bancilhon and Ramakrishnan (1988) and Greco (2003) basically resort to the execution of some recursive queries on a variety of data structures. The main goal of our experiments was to evaluate the deductive capabilities of tested systems for both query answering time and amount of manageable data, especially with respect to the direct database execution of our system.

All tests have been carried out on a Pentium 4 processor with a 1.4-GHz CPU and 512 Mbytes of RAM.

[8] It incorporates the Parser, the Optimizer, and the DG Builder depicted in Figure 6.

[9] As previously pointed out, the presence of an #export command automatically limits the system to generate the first answer set only.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

6.1 Overview of compared systems

To provide a comparative and comprehensive analysis with the state-of-the-art systems in the considered research area, we have compared our system performance, under both execution modalities (i.e, DLV^{DB} and DLV^{IO}), with (i) LDL++, because it is one of the most robust implementations of DDSs; (ii) XSB, as an efficient implementation of the top-down evaluation strategy; (iii) Smodels, one of the most widely used Answer Set Programming systems together with DLV; and (iv) three commercial DBMSs supporting the execution of recursive queries. Note that the licence of use of such DBMSs does not allow us to explicitly mention them in the article; as a consequence, in the following, we call them simply DB-A, DB-B, and DB-C. The reader should just know they are the three top-level commercial database systems currently available, which also support recursive queries.

Note that important DBMSs such as Postgres and MySQL could not be tested; in fact, they do not support recursive queries, which are the basis for our testing framework. Moreover, as we pointed out in the Introduction, other logic-based systems such as ASSAT, Cmodels, and CLASP have not been tested since they use the same grounding layer of Smodels (LParse) and, as it will be clear in the following, the benchmark programs are completely solved by this layer.

In the following, we briefly overview the main characteristics of the tested systems, focusing on their support to the language and technological capabilities addressed in this work. Specifically, we consider, for each database system, its capability to express recursive queries and, for each logic-based system, the expressiveness of its language and its capability to interact with external DBMSs.

For each system, we used the latest release available at the time tests have been carried out.

6.1.1 Database systems

As far as database systems are concerned, it is worth pointing out that none of the considered ones fully adopt the SQL99 standard for the definition of recursive queries, but proprietary constructs are introduced by each of them.

In particular, both DB-A and DB-B support the standard recursive functionalities that are needed for our benchmarks, even if proprietary constructs must be added to the standard SQL99 statement to guarantee the termination of some kinds of queries. On the contrary, DB-C implements a large subset of SQL99 features and supports recursion, but, as far as recursive queries are concerned, it exploits proprietary constructs that do not follow the standard SQL99 notation, and whose expressiveness is lower than that of SQL99; as an example, it is not possible to express unbound queries within recursive statements (e.g., *all* the pairs of nodes linked by at least one path in a graph).

6.1.2 LDL++

The LDL++ system (Arni *et al.* 2003) integrates rule-based programming with efficient secondary memory access, transaction management recovery, and integrity

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

control. The underlying database engine has been developed specifically within the LDL project and is designed as a virtual-memory record manager, which is optimized for the situation where the pages containing frequently used data can reside in main-memory. LDL++ can also be interfaced with external DBMSs, but it is necessary to implement vendor-specific drivers to handle data conversion and local SQL dialects (Arni *et al.* 2003). The LDL++ language supports complex terms within facts and rules, stratified negation, and do not care non-determinism based on stable model semantics. Moreover, LDL++ supports updates through special rules.

In our tests, we used version 5.3 of LDL++. Test data have been fed to the system by text files storing input facts.

6.1.3 XSB

The XSB system (Rao *et al.* 1997) is an in-memory deductive database engine based on a Prolog/SLD resolution strategy called SLG. It supports explicitly locally stratified programs. The inference engine, which is called SLG-WAM, consists of an efficient tabling engine for definite logic programs, which is extended by mechanisms for handling cycles through negation. These mechanisms are negative loop detection, delay, and simplification. They serve for detecting, breaking, and resolving cycles through negation.

XSB allows the exploitation of data residing in external databases, but reasoning on such data is carried out in main-memory. In our tests, we have used version 2.2 of XSB.

6.1.4 SModels

The SModels system (Niemelä and Simons 1997; Niemelä *et al.* 2000) implements the answer set semantics for normal logic programs extended by built-in functions as well as cardinality and weight constraints for domain-restricted programs.

The SModels system takes as input logic program rules in Prolog style syntax. However, in order to support efficient implementation techniques and extensions, the programs are required to be *domain-restricted* where the idea is the following: the predicate symbols in the program are divided into two classes, *domain predicates* and *non-domain* predicates. Domain predicates are predicates that are defined nonrecursively. The main intuition of domain predicates is that they are used to define the set of terms over which the variable range in each rule of a program P. All rules of P have to be domain-restricted in the sense that every variable in a rule must appear in a domain predicate that appears positively in the rule body. In addition to normal logic program rules, SModels supports rules with cardinality and weight constraints, which are similar to `#count` and `#sum` aggregates of DLV.

SModels does not allow to handle data residing in database relations; moreover, all the stages of the computation are carried out in main-memory. Finally, it does not support optimization strategies for bound queries; consequently, the time it needs for executing the same query either with all parameters unbound or with some parameters bound is exactly the same.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

In our tests, we used SModels, version 2.28 with LParse version 1.0.17. Test data have been fed to the system by text files storing input facts.

6.1.5 DLV^{DB}

It is the direct database execution of our system. In our tests, we used a commercial database as DBMS for the working database. However, to guarantee fairness with the other systems, we did not set any additional index or key information for the involved relations. We point out again that any DBMS supporting ODBC could be easily coupled with DLV^{DB}.

6.1.6 DLV^{IO}

It is the main-memory execution modality of the system presented in this article. Recall that it basically corresponds to the execution of the standard DLV system with data loaded from databases.

6.2 Benchmark problems

To asses the performance of the systems described above, we carried out several tests using classical benchmark problems from the context of deductive databases (Bancilhon and Ramakrishnan, 1988; Greco, 2003), namely, *Reachability* and *Same Generation*. The former allows the analysis of basic recursion capabilities of the various systems on several data structures, whereas the latter implements a more complex problem and, consequently, allows the capability of the considered systems to carry out more refined reasoning tasks to be tested.

For each problem, we measured the performance of the various systems in computing three kinds of queries, namely: unbound queries (identified by the symbol $\mathcal{Q}_0$ in the following); and queries with one bound parameter ($\mathcal{Q}_1$); and queries with all bound parameters ($\mathcal{Q}_2$). Considering these three cases is important because DBMSs and deductive databases generally benefit from query bindings (by "pushing down" selections through relational algebra optimizations, magic set techniques, or, for XSB, top-down evaluation), whereas ASP systems are generally more effective with unbound queries (since they usually compute the entire models anyway); as a consequence, it is interesting to test all these systems in both their favorable and unfavorable contexts. It is worth pointing out that some of the tested systems implement optimization strategies "a la magic set" (Bancilhon *et al.* 1986; Ross 1990; Beeri and Ramakrisnhan 1991; Mumick *et al.* 1996) (e.g., DLV^{DB} and LDL++), typical of deductive databases, or other program rewriting techniques. As a consequence, the actually evaluated programs are the optimized ones automatically derived by these systems, but the cost of these rewritings has been always considered in the measure of systems' performance.

In what follows, we briefly introduce the two considered problems; the interested reader can find all details about them in Bancilhon and Ramakrishnan (1988).

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

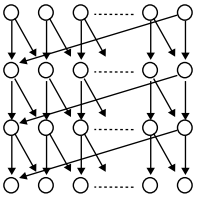

Fig. 8. Example of a cylinder graph.

6.2.1 Reachability

Given a directed graph $G = (V, E)$, the solution to the reachability problem *reachable*(a, b) determines whether a node $b \in V$ is reachable from a node $a \in V$ through a sequence of edges in E. The input is provided by a relation *edge*(X, Y), where a fact *edge*(a, b) states that b is directly reachable by an edge from a.

In database terms, determining all pairs of reachable nodes in G amounts to computing the transitive closure of the relation storing the edges.

6.2.2 Same generation

Given a parent-child relationship (a tree), the Same Generation problem aims to find pairs of persons belonging to the same generation. Two persons belong to the same generation either if they are siblings or if they are children of two persons of the same generation.

The input is provided by a relation *parent*(X, Y), where a fact *parent*(*thomas*, *moritz*) means that *thomas* is the parent of *moritz*.

6.3 Benchmark data sets

For each considered problem, we exploited several sets of benchmark data structures. For each data structure, various instances of increasing dimensions have been constructed; the size of each instance is measured in terms of the number of input facts describing it.

6.3.1 Reachability

As for the Reachability problem, we considered (i) full binary trees, (ii) acyclic graphs (a-graphs in the following), (iii) cyclic graphs (c-graphs in the following), and (iv) cylinders (Bancilhon and Ramakrishnan 1988).

The *density* δ of a graph can be measured as $\delta = \frac{\#\text{ of arcs in the graph}}{\#\text{ of possible arcs}}$. We generated various typologies of graph instances, characterized by values of δ equal to 0.20, 0.50, and 0.75 respectively. Because of space constraints, in this article, we report just the results obtained for $\delta = 0.20$.

Cylinders are particular kinds of acyclic graphs that can be layered; each layer has the same number of nodes. Each node of the first layer has two outgoing arcs and no incoming arcs, whereas each node of the last layer has two incoming arcs and no outgoing arcs; finally, each node of an internal layer has two incoming and two outgoing arcs. An example of a cylinder is shown in Figure 8. A cylinder has

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

then a width and a height; as a consequence, the ratio $\rho = \frac{\text{width}}{\text{height}}$ can be exploited to characterize a cylinder. We generated various categories of cylinders having ρ equal to 0.5, 1.0, and 1.5, respectively. Because of space constraints, in this article, we report just the results obtained for $\rho = 1$.

Graphs have been generated using the Stanford GraphBase (Knuth 1994) library, whereas trees and cylinders have been generated using ad hoc procedures, since they are characterized by a regular structure.

6.3.2 Same generation

As far as the Same Generation problem is concerned, we exploited full binary trees as input data structures.

6.4 Problem encodings

We have used general encodings for the two considered problems in a way that tests the various systems under generic conditions; specifically, we used "uniform" queries, that is, queries whose structure must not be modified depending on the quantity and positions of bound parameters. Several alternative encodings could have been possible for the various problems, depending also on the underlying data structures. However, since many other problems of practical relevance can be brought back to the ones we considered, we preferred to exploit those encodings applicable to the widest variety of applications.

Because of space constraints, we cannot list here the encodings exploited in our tests. The interested reader can find them at
`http://www.mat.unical.it/terracina/tplp-dlvdb/encodings.pdf`.

Note that, since DB-C does not support the standard SQL99 language but only a simplified form of recursion, we have not tested this system along with the other ones. We discuss encodings and results obtained for DB-C in a separate section.

6.5 Results and discussion

In our tests, we measured the time required by each system to answer the various queries. We fixed a maximum running time of 12,000 s (about 3 h) for each test. In the following figures, the line of a system stops whenever some query was not solved within this time limit (note that graphs have a logarithmic scale on the vertical axis).

In more detail, Figures 9–11 show results obtained for the various tests; the headline of each graph reports the corresponding query.

From the analysis of these figures, we can observe that, in several cases, the performance of DLV^{DB} (the black triangle in the graphs) is better than all the other systems with orders of magnitude and that DLV^{DB} allows almost always the handling of the greatest amount of data; moreover, there is no system that can be considered the "competitor" of DLV^{DB} in *all* the tests.

In particular, in some tests, XSB shows a good behavior (e.g., in Reachability on cyclic graphs and cylinders) but, even in those positive tests, it "dies" earlier

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

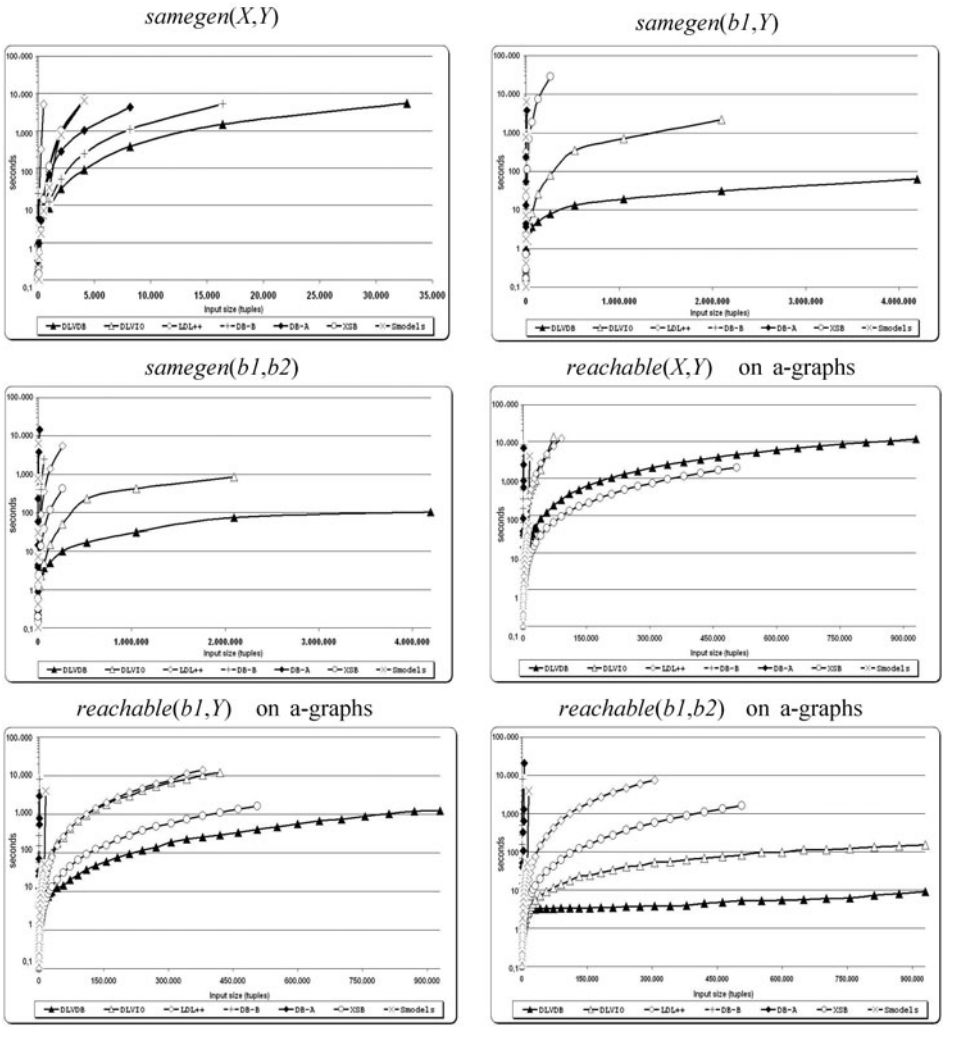

Fig. 9. Results for Same Generation on trees and Reachability with acyclic graphs.

than DLV^{DB} (with the exception of *reachable*(*b1*,*Y*) on cylinders), probably because it exceeds the main-memory.

LDL++ is competitive with DLV^{DB} only in *reachable*(*b1*,*Y*) on cyclic graphs and cylinders, whereas in all the other queries, the performance difference is of more than one order of magnitude.

DB-B performance is near to that of DLV^{DB} only in *samegen*(*X*,*Y*); in all the other cases, its line is near to the vertical axis.

DB-A showed very good performance only for reachability on trees (see also Table 1 introduced next). This behavior could be justified by the presence of optimization mechanisms implemented in this system that are particularly suited for computing the transitive closure on simple data structures (such as trees), but these are not effective for other (more complex) kinds of query/data structure.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

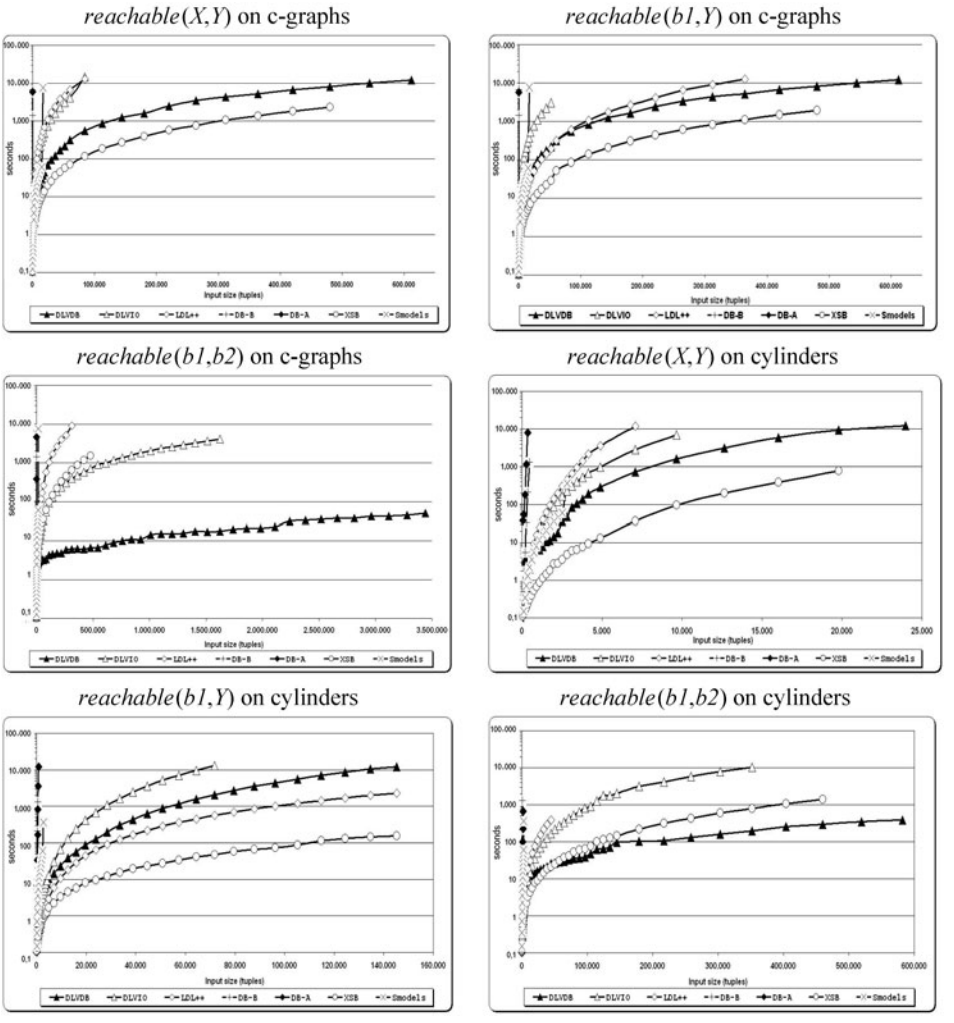

Fig. 10. Results for Reachability with cyclic graphs and with cylinders.

Surprisingly enough, DBMSs often have the worst performance (their times are near to the vertical axis) and they can handle very limited amounts of input data.

Finally, as expected, DLV^{IO} is capable of handling lower amounts of data with respect to DLV^{DB}; however, in several cases it was one of the best three performing systems, especially on bound queries. This result is mainly due to the magic sets optimization technique it implements.

A rather surprising result is that DLV^{IO} has almost always higher execution times than DLV^{DB} even for not very high input data sizes. The motivation for this result can be justified by the following reasoning. Both DLV^{DB} and DLV^{IO} benefit from all the program rewriting optimization techniques developed in the DLV project. Moreover, both of them implement a differential Semi-Naive approach for the evaluation of normal stratified programs. However, while DLV^{IO} reasons about its underlying data in a tuple-at-a-time way, DLV^{DB} exploits a set-at-a-time strategy

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

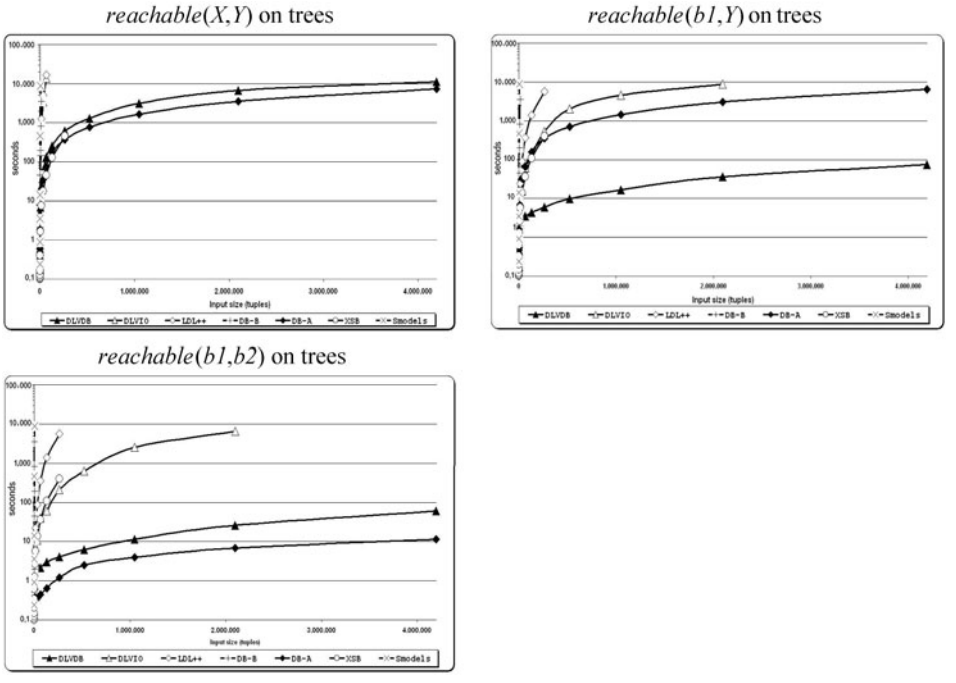

Fig. 11. Results for Reachability with trees.

(implemented by SQL queries); this, in conjunction with the fact that the underlying working database implements advanced data-oriented optimization strategies, makes DLV^{DB} more efficient than DLV^{IO} even when all the data fit in main-memory.

Similarly, as pointed out in Bancilhon and Ramakrishnan (1988), another important parameter to measure in this context is the system's capability of handling large amounts of data. To carry out this verification, we considered the time response of each system for the largest input data set we have used in each query.

Table 1 shows the execution times measured for those systems that have been capable of solving the query within the fixed time limit of 12,000 s; the second column of the table shows, for each query, both the input data size, measured in terms of the number of input facts (tuples), and the total amount of handled data, measured in Mbytes, given by the size of the answer set produced by DLV^{DB} in answering that query.[10]

From the analysis of this table, we may observe that (i) DLV^{DB} has been always capable of solving the query on the maximum data size; (ii) in 11 queries out of 15 DLV^{DB} (in one case along with DLV^{IO}) has been the only system capable of completing the computation within the time limit; (iii) DLV^{DB} allowed to handle up to 6.7 Gbytes of data in *samegen*(X,Y) and 1.6 Gbytes in *reachable*(X,Y) on trees within the fixed time limit of 12,000 s and never ended its computation owing to lack of memory, as other systems did.

[10] Note that all facts produced by DLV^{DB} to answer the query are considered.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

Table 1. *Execution times of the systems capable of solving the query for the maximum considered size of the input data*

Query / Data type	Input size (tuples) / Output size (Mbytes)	DB-B (s)	DLV^{IO} (s)	DLV^{DB} (s)	LDL++ (s)	SModels (s)	DB-A (s)	XSB (s)
samegen(X,Y)	32,766	···	···	5,552	···	···	···	···
tree	*6,716 Mb*							
samegen(b1,Y)	4,194,302	···	···	64	···	···	···	···
tree	*78 Mb*							
samegen(b1,b2)	4,194,302	···	···	102	···	···	···	···
tree	*78 Mb*							
reachable(X,Y)	929,945	···	···	11,820	···	···	···	···
a-graph	*103 Mb*							
reachable(b1,Y)	929,945	···	···	1,191	···	···	···	···
a-graph	*38 Mb*							
reachable(b1,b2)	929,945	···	···	4	···	···	···	···
a-graph	*17 Mb*							
reachable(X,Y)	612,150	···	···	11,936	···	···	···	···
c-graph	*68 Mb*							
reachable(b1,Y)	612,150	···	···	11,933	···	···	···	···
c-graph	*68 Mb*							
reachable(b1,b2)	612,150	···	981	8	···	···	···	···
c-graph	*11 Mb*							
reachable(X,Y)	23,980	···	···	11,784	···	···	···	···
cylinder	*465 Mb*							
reachable(b1,Y)	145,260	···	···	11,654	2284	···	···	157
cylinder	*279 Mb*							
reachable(b1,b2)	582,120	···	···	388	···	···	···	···
cylinder	*13 Mb*							
reachable(X,Y)	4,194,302	···	···	11,161	···	···	7,280	···
tree	*1,634Mb*							
reachable(b1,Y)	4,194,302	···	···	76	···	···	6,438	···
tree	*79 Mb*							
reachable(b1,b2)	4,194,302	···	···	60	···	···	12	···
tree	*78 Mb*							

6.6 Comparison to DB-C

As previously pointed out, DB-C does not support the standard SQL99 encoding for recursive queries, but it exploits a proprietary language for implementing a simplified form of recursion. This language is less expressive than SQL99 for recursion; as an example, unbound recursive queries cannot be implemented in DB-C; analogously, it does not allow to write recursive views in a "uniform" way (i.e., independently from the specific bound parameters).

As for the problems addressed in this article, it was not possible to write the unbound query either for Reachability or for Same Generation with DB-C. The other queries have encodings not equivalent to the general version we adopted for the other systems.

As an example, the query $\mathcal{Q}_1 = reachable(b1,Y)$ can be expressed in DB-C by the following statement:

SELECT $b1$, $edge.att_2$ FROM edge
START WITH $att_1 = b1$ CONNECT BY PRIOR $att_2 = att_1$

which, however, is equivalent to the datalog program:

$reached(b1).$
$reached(X) \text{ :- } reached(Y),\ edge(Y,X).$
$reachable(b1,Y) \text{ :- } reached(Y).$

This is clearly a program that can be evaluated more easily than the general encoding, because it involves a recursive rule with one single attribute and a unique

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

starting point for the recursion (the fact *reached(b1)*). However, this query (and the equivalent program) is less general than the one introduced in Section 6.4, since its structure must be modified if, for example, we need to have both the parameters bound or we want to bound the second parameter instead of the first.

Clearly, testing such encodings against the other, more general, ones would have been unfair. Anyway, we carried out some tests involving DB-C, by applying its encodings and the corresponding datalog programs on the maximum data instances we considered for the various queries, in order to have a rough idea on the performance. As an example, for the query $\mathcal{Q}_1 = reachable(b1,Y)$ mentioned above, on a-graphs (resp., c-graphs) of size 929,945 (resp., 612150) tuples we have measured that DB-C takes 22.5 (resp., 15.9) s, whereas DLV^{DB} takes 6.4 (resp., 5.6) s. Analogously, for the query $\mathcal{Q}_1 = samegen(b1,Y)$, on trees of size 4,194,302 tuples, DB-C requires 1,329.4 s to terminate the computation, whereas DLV^{DB} requires 500.8 s. DB-C performed better than DLV^{DB} only for Reachability on trees. In addition, as we have done for DB-A, we may conjecture that this behavior is motivated by the particular optimization techniques implemented in the system.

These results are representative of the overall performance we have measured for DB-C in our benchmarks. On the one hand, they confirm our claim that the encodings solvable by DB-C are very different, also from a performance point of view, with respect to the general ones used in our benchmarks (as an example, this is proved by the significantly lower timing measured for DLV^{DB} in $reachable(b1,Y)$ with respect to the same query in the standard encoding). On the other hand, they allow us to conclude that the same reasoning as that drawn in Section 6.5 about DLV^{DB} performance is still valid.

7 Conclusions

In this article, we have presented DLV^{DB}, a new deductive system for reasoning on massive amounts of data. It not only presents features of an efficient DDS but also extends the capability of handling data residing in external databases to a DLP system. A thorough experimental validation showed that DLV^{DB} provides both important speed ups in the running time of typical deductive queries and the capability to handle larger amounts of data with respect to existing systems. Interestingly, the experimental results show that DLV^{DB} significantly outperforms both commercial DBMSs and other logic-based systems in the evaluation of recursive queries.

The key reason for the relevant performance improvement obtained by our system is the integration of the following factors: (i) The idea to employ the efficient engine of a commercial DBMS for rule evaluation, by translating logical rules in SQL statements (which are then executed by a DBMS), thus allowing us to exploit the efficient data-oriented optimization techniques of relational databases. (ii) The exploitation of advanced optimization techniques developed in the field of deductive databases for logical query optimization (e.g., magic sets). (iii) A proper combination and a well-engineered implementation of the above ideas. Moreover, the usage of a purely mass-memory evaluation strategy improves previous deductive systems, eliminating, in practice, any limitation in the dimension of the input data.

https://doi.org/10.1017/S1471068407003158 Published online by Cambridge University Press

In the future, we plan to extend the language supported by the direct database execution and to exploit the system in interesting research fields such as data integration and data warehousing. Moreover, a mixed approach exploiting both DLV^{DB} and DLV^{IO} executions to evaluate difficult problems partially in mass-memory and partially in main-memory will be explored.

Acknowledgments

This work was partially supported by the Italian "Ministero delle Attività Produttive" under project "Discovery Farm" B01/0297/P 42749-13, and by M.I.U.R. under project "ONTO-DLV: Un ambiente basato sulla Programmazione Logica Disgiuntiva per il trattamento di Ontologie" 2521.

References

American National Standards Institute. 1999. *ANSI/ISO/IEC 9075-1999 (SQL:1999, Parts 1–5)*. American National Standards Institute, New York.

Abiteboul, S., Abrams, Z., Haar, S. and Milo, T. 2005. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS, 2005)*, Baltimore, MD, ACM, New York, NY, USA. 358–367.

Apt, K. R., Blair, H. A. and Walker, A. 1988. Towards a Theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, Washington, DC, 89–148.

Arni, F., Ong, K., Tsur, S., Wang, H. and Zaniolo, C. 2003. The Deductive Database System LDL++. *Journal of the Theory and Practice of Logic Programming 3,* 1, 61–94.

Balbin, I. and Ramamohanarao, K. 1987. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programing 4,* 3, 259–262.

Bancilhon, F., Maier, F., Sagiv, Y. and Ullman, J. 1986. Magic sets and other strange ways to implement logic programs. In *Proc. of the ACM Symposium on Principles of Database Systems (PODS'86)*. ACM Press, Cambridge, MA, 1–16.

Bancilhon, F. and Ramakrishnan, R. 1988. Performance evaluation of data intensive logic programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Washington, DC, 439–517.

Baral, C. 2002. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, New York.

Beeri, C. and Ramakrisnhan, R. 1991. On the power of magic. *Journal of Logic Programming 10,* 1–4, 255–259.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12*, 53–87.

Ben-Eliyahu, R. and Dechter, R. 1996. On computing minimal models. *Annals of Mathematics and Artificial Intelligence 18,* 1, 3–27.

Ceri, S., Gottlob, G. and Tanca., L. 1990. *Logic Programming and Databases*. Springer Verlag, New York.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N. and Pfeifer, G. 2003a. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, Acapulco, Mexico. Morgan Kaufmann, Washington, DC, 847–852.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N. and Pfeifer, G. 2003b. Aggregate Functions in DLV. In *Proceedings ASP03—Answer Set Programming: Advances in Theory and Implementation*, Messina, Italy, M. de Vos and A. Provetti, Eds. 274–288. URL: http://CEUR-WS.org/Vol-78/. Access date: 30/10/2006.

Faber, W., Leone, N., Mateis, C. and Pfeifer, G. 1999a. Using database optimization techniques for nonmonotonic reasoning. In *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, I. O. Committee, Ed. Prolog Association of Japan, 135–139.

Faber, W., Leone, N. and Pfeifer, G. 1999b. Pushing goal derivation in DLP computations. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, El Paso, TX, M. Gelfond, N Leone, and G. Pfeifer, Eds. Number 1730 in Lecture Notes in AI (LNAI). Springer Verlag, New York, 177–191.

Faber, W., Leone, N. and Pfeifer, G. 2001. Experimenting with heuristics for answer set programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, Seattle, WA, Morgan Kaufmann, Washington, DC, 635–640.

Faber, W., Leone, N. and Pfeifer, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In Proc. of JELIA 2004. 200–212.

Faber, W. and Pfeifer, G. 1996. DLV homepage. URL: http://www.dlvsystem.com/. Access date: 30/10/2006.

Gallaire, H., Minker, J. and Nicolas, J. 1984. Logic and databases: A deductive approach. *ACM Computing Surveys 16,2*, 153–186.

Garcia-Molina, H., Ullman, J. D. and Widom, J. 2000. *Database System Implementation*. Prentice Hall, Upper Soddle River, NJ.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI) 2007*, Hyderabad, India. AAAI Press, Menlo Park CA, 386–392.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Giunchiglia, E., Lierler, Y. and Maratea, M. 2006. Answer Set Programming based on Propositional Satisfiability. *Journal of Automated Reasoning (JAR) 36*, 4, 345–377.

Giunchiglia, E., Maratea, M. and Lierler, Y. 2004. SAT-based answer set programming. In *American Association for Artificial Intelligence*. AAAI Press, Menlo Park, CA, 61–66.

Grant, J. and Minker, J. 1992. The impact of logic programming on databases. *Communications of the ACM 35*, 3, 66–81.

Greco, S. 2003. Binding propagation techniques for the optimization of bound disjunctive queries. *IEEE Transactions on Knowledge and Data Engineering 15,* 2, 368–385.

Knuth, D. E. 1994. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. *The DLV System for Knowledge Representation and Reasoning*. ACM Transactions on Computational Logic, ACM Press, New York. URL: http://www.arxiv.org/ps/cs.AI/0211004. Access date: 30/10/2006.

Lin, F. and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *American Association for Artificial Intelligence*. AAAI Press, Menlo Park, CA, 112–118.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157,* 1–2, 115–137.

Loo, B., Hellerstein, J., Stoica, I. and Ramakrishnan, R. 2005. Declarative routing: extensible routing with declarative queries. In *Proceedings of the ACM SIGCOMM*

2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, PA, ACM press, New York, NY, USA, 289–300.

Lu, J., Nerode, A. and Subrahmanian, V. 1996. Hybrid knowledge bases. *IEEE Transactions on Knowledge and Data Engineering 8,* 5, 773–785.

Mumick, I., Finkelstein, S., Pirahesh, H. and Ramakrishnan, R. 1996. Magic conditions. *ACM Transactions on Database Systems 21*, 1, 107–155.

Niemelä, I. and Simons, P. 1997. 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)*, Dagstuhl, Germany. J. Dix, U. Furbach and A. Nerode, Eds. Lecture Notes in AI (LNAI), vol. 1265. Springer Verlag, New York, 420–429.

Niemelä, I., Simons, P. and Syrjänen, T. 2000. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, CO, C. Baral and M. Truszczyński, Eds. Electronic proceedings; available at: `http://xxx.lanl.gov/abs/cs.AI/0003033`. Access date: 30/10/2006.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, Washington, DC, 193–216.

Rao, P., Sagonas, K., Swift, T., Warren, D. and Friere, J. 1997. XSB: A system for efficiently computing well-founded semantics. In *Proc. of 4th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR'97)*. Springer, LNAI, New York, 430–440.

Ross, K. 1990. Modular stratification and magic sets for datalog programs with negation. In *Proc. of the ACM Symposium on Principles of Database Systems*. Nashville, Tennessee. ACM Press, New York, 161–171.

Ullman, J. 1989. *Principles of Database and Knowledge Base Systems*. Computer Science Press. New York, NY, USA.

Winslett, M. 2006. Raghu Ramakrishnan speaks out. *SIGMOD Record 35,* 2, 77–85.

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S. and Zicari, R. 1997. *Advanced Database Systems*. Morgan Kaufmann, Washington, DC.