Document Number:	D1.2
Project Name:	Infomix: Boosting the Information Integration
Project Number:	IST-2001-33570
Title:	**Review of methods and systems from computational logic**
Authors:	Nicola Leone, Pasquale Rullo
Workpackage:	WP1
Document Type:	Deliverable
Classification:	Official
Distribution:	Public
Status:	Final
Document file:	
Version:	1.0
Date:	July 31, 2002
Number of pages:	49

Document Change Record		
Version	**Date**	**Reason for Change**

Contents

1 Introduction

Information integration [8, 53, 90, 97] is the task of providing a uniform interface to various pre-existing data sources, so as to enable users to focus on specifying what they want, rather than thinking of how to obtain the answer. As a result, an information integration system frees the users from the task of finding the relevant data sources, interacting with each source in isolation, and selecting, cleaning, and combining data from multiple sources.

Information integration is a highly complex problem, comprising aspects ranging from modelling to query processing, query optimization, inconsistency handling etc. Exploiting computational logic systems for solving reasoning tasks in such complex scenario seems to be a natural approach.

In the recent years, research in computational logic has produced a number of implemented systems [66, 32, 23, 82, 75, 91, 41, 47, 96, 76, 18, 23, 73, 12, 70, 69, 17, 24, 27] by which various advanced reasoning problems such as diagnosis, configuration, etc. can be declaratively solved in logic-based languages. The underlying computational engines have been developed with quite some effort, and comprise a body of sophisticated tools and algorithms.

This report is devoted to give a review of methods and systems from computational logic and to point out the applicability of them to data integration tasks. In particular, since our goal is to integrate computational logic systems in a data integration framework, in this report, we focus on computational logic systems which can be seen as extensions of Database Systems. In particular, we first focus on deductive database systems [35, 16, 40] which, essentially, extend relational database systems [64] by recursion and stratified negation. Then, we analyze the more recent answer set programming systems [55, 27] which go beyond deductive databases, supporting also unstratified negation and other advanced constructs like disjunction, and various forms of constraints. These systems are based on a bottom-up computational model which is closer to the computational model of database systems than the top-down (proof theoretic) computational model, and it is better suited for dealing with larger amounts of data. The above class of computational logic systems (including both deductive database systems and answer set programming systems) is often referred to as the class of *declarative* computational logic systems, since these systems support a fully declarative programming style (while in Prolog the result depends on the ordering of the rules in the program, and also on the ordering of the goals in the bodies of the rules).

The report is organized as follows. In Section 2 we define the language of answer set programs (a logic programming language extending the language of deductive databases). In Section 3 we deal with Knowledge Representation methods in computational logic; in particular, we show the way how queries and, more in general, search problems can be encoded through answer set programs. Features of computational logic systems relevant to data integration activities are pointed out in Section 4. These features are exploited throughout the report to characterize the computational logic systems under considerations from the veiwpoint of data integration. In Section 5 we describe the most relevant (declarative) computational logic systems. In particular, in Section 5.1 we overview the most popular deductive database systems; while answer set systems are discussed in Section 5.2. The overview of Section 5 points out that answer set systems are better suited than deductive database systems for data integration, most importantly because the latter miss the computational power to solve some hard (CONP-complete) tasks arising in data-integration. Therefore, in Section 6 we deepen the analysis of answer set programming systems, by performing also a set of experiments to compare the efficiency of the most popular answer set systems, namely **DLV** and SMODELS. Finally, in Section 7 we draw our conclusions and make our recommendations.

2 Language

In this section, we provide a formal definition of syntax and semantics of the language of answer set programs (which is a superset of the language of deductive database systems): disjunctive datalog under the consistent answer sets semantics extended with strong negation and weak constraints. For further background, see [61, 28, 38].

2.1 Syntax

Following Prolog's convention, strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. A *term* is either a variable or a constant. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *classical literal* l is either an atom p (in this case, it is *positive*), or a negated atom $\neg p$ (in this case, it is *negative*). A *negation as failure (NAF) literal* ℓ is of the form l or `not` l, where l is a classical literal; in the former case, it is *positive*, and in the latter case *negative*. Unless stated otherwise, by *literal* we mean a classical literal.

Given a classical literal l, its *complementary* literal is defined as $\neg p$ if $l = p$ and p if $l = \neg p$. A set L of literals is said to be *consistent* if for every literal $l \in L$, its complementary literal is not contained in L.

In addition to literals as defined above, disjunctive datalog programs also support built-ins like equality, less-than, and greater-than, as well as various arithmetic predicates like addition or multiplication.

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \ \texttt{v} \ \cdots \ \texttt{v} \ a_n \ \texttt{:-} \ b_1, \cdots, b_k, \ \texttt{not} \ b_{k+1}, \cdots, \ \texttt{not} \ b_m. \tag{1}$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \texttt{v} \ \cdots \ \texttt{v} \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \ \texttt{not} \ b_{k+1}, ..., \ \texttt{not} \ b_m$ is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), we usually omit the " `:-` " sign.

The following notation will be useful for further discussion. A set $\{\texttt{not} \ l_i, \ldots, \texttt{not} \ l_{i+k}\}$ will be denoted by $\texttt{not}\{l_i, \ldots, l_{i+k}\}$. If r is a rule of type (1) then $H(r) = \{a_1, \ldots, a_n\}$ is the set of literals in the head, $B(r) = B^+(r) \cup B^-(r)$ is the set of the body literals, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$.

A *disjunctive datalog program* $\mathcal{P}$ is a finite set of rules. A `not`-free program $\mathcal{P}$ (i.e. $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive program*, and a `v`-free program $\mathcal{P}$ (i.e. $\forall r \in$

$\mathcal{P} : |H(r)| \leq 1)$ is called *datalog program.* A disjunctive datalog program is also called *disjunctive logic program* and *disjunctive deductive database.* A datalog program is also called *normal logic program* and *deductive database.*

Some languages for disjunctive datalog programs extend disjunctive datalog by another construct, which is called *weak constraint* [13]. We define weak constraints as variants of integrity constraints. In order to differentiate clearly between these two, we use the symbol :∼ instead of `:-`. Additionally, a weight and a priority level or layer (inducing a partial order) of the weak constraint are specified explicitly.

Formally, a weak constraint w is an expression of the form

$$:\sim \; b_1, \ldots, b_k, \; \texttt{not} \; b_{k+1}, \ldots, \; \texttt{not} \; b_m.[w : l]$$

where for $m \geq k \geq 0$, the $b_1, \ldots, b_m$ are classical literals, while w (the *weight*) and l (the *level* or *layer*) are positive integers or variables. For convenience, w and/or l might be omitted and are set to 1 by default. The sets $B(w)$, $B^+(w)$, and $B^-(w)$ of a weak constraint w are defined as for regular integrity constraints.

In what follows, we denote by DLP^w the language of disjunctive datalog extended with weak constraints.

A DLP^w program $\mathcal{P}$ (for short, simply *program*) is a finite set of rules[1] and weak constraints. In other words, a DLP^w program $\mathcal{P}$ is a disjunctive datalog program with weak constraints. For a program $\mathcal{P}$, let $WC(\mathcal{P})$ denote the set of weak constraints in $\mathcal{P}$, and $Rules(\mathcal{P})$ denote the set of rules (including integrity constraints) in $\mathcal{P}$.

As usual, a term (an atom, a rule,...) is called *ground* if no variables appear in it. A ground program is also called a *propositional* program.

2.2 Semantics

The semantics of DLP^w programs extends the consistent answer sets semantics of disjunctive datalog programs, originally defined in [38], to deal with weak constraints. The semantics provided in this section is a slight generalization of the original semantics proposed for weak constraints in [13]. In particular, the language proposed in [13] allowed only "prioritized weak constraints"; while the DLP^w language allows both priority levels (layers) and weights for weak constraints.

Herbrand Universe Given a program $\mathcal{P}$, let $U_{\mathcal{P}}$ (the Herbrand Universe) be the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant χ is added to $U_{\mathcal{P}}$.

Herbrand Literal Base Given a program $\mathcal{P}$, let $B_{\mathcal{P}}$ be the set of all ground (classical) literals constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_{\mathcal{P}}$ (note that $B_{\mathcal{P}}$ contains also the atoms preceded by the strong negation symbol $\neg$).

[1] Recall that an integrity constraint is a rule as well.

Ground Instantiation Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. In a similar way, given a weak constraint w, $Ground(w)$ denotes the set of weak constraints obtained by applying all possible substitutions σ from the variables in w to elements of $U_{\mathcal{P}}$. Given a program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $GroundRules(\mathcal{P}) \cup GroundWC(\mathcal{P})$, where $GroundRules(\mathcal{P}) = \bigcup_{r \in Rules(\mathcal{P})} Ground(r)$ and $GroundWC(\mathcal{P}) = \bigcup_{w \in WC(\mathcal{P})} Ground(w)$. For propositional programs, $\mathcal{P} = Ground(\mathcal{P})$ holds.

Answer Sets For every program $\mathcal{P}$, we define its answer sets using its ground instantiation $Ground(\mathcal{P})$ in three steps: First we define the answer sets of positive disjunctive datalog programs, then we give a reduction of disjunctive datalog programs containing negation as failure to positive ones and use it to define answer sets of arbitrary disjunctive datalog programs, possibly containing negation as failure. Finally, we specify the way how weak constraints affect the semantics, defining the semantics of general DLP^w programs.

An interpretation I is a set of classical literals. A consistent interpretation $X \subseteq B_{\mathcal{P}}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive disjunctive datalog program), if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. A set $X \subseteq B_{\mathcal{P}}$ is an *answer set* for positive disjunctive datalog program $\mathcal{P}$ if it is a minimal set (w.r.t. set inclusion) that is closed under $\mathcal{P}$.

Example 2.1 For the positive program $\mathcal{P}_1 = \{a \texttt{ v } \neg b \texttt{ v } c.\}$, the answer sets are $\{\texttt{a}\}$, $\{\neg\texttt{b}\}$, and $\{\texttt{c}\}$.

For the positive program $\mathcal{P}_2 = \{a \texttt{ v } \neg b \texttt{ v } c. \;\; ; \;\; \texttt{:-}\, a.\}$, the sets $\{\neg\texttt{b}\}$ and $\{\texttt{c}\}$ are the answer sets.

Finally, for the positive program $\mathcal{P}_3 = \{a \texttt{ v } \neg b \texttt{ v } c. \;\; ; \;\; \texttt{:-}\, a. \;\; ; \;\; \neg b \texttt{ :- } c. \;\; ; \;\; c \texttt{ :- } \neg b.\}$, the set $\{\neg\texttt{b}, \texttt{c}\}$ is the only answer set.

The *reduct* or *Gelfond-Lifschitz transform* of a ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by

- deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds;
- deleting the negative body (i.e., $B^-(r)$) from the remaining rules.

An answer set of a disjunctive datalog program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

Example 2.2 Given the program $\mathcal{P}_4 = \{a \texttt{ v } \neg b \texttt{ :- } c. \; ; \; \neg b \texttt{ :- not } a, \texttt{not } c. \; ; \; a \texttt{ v } c \texttt{ :- not } \neg b.\}$ and $I = \{\neg\texttt{b}\}$, the reduct $\mathcal{P}_4^I$ is $\{a \texttt{ v } \neg b \texttt{ :- } c. \;\; ; \;\; \neg b.\}$. It is easy to see that I is an answer set of $\mathcal{P}_4^I$, and for this reason it is also an answer set of $\mathcal{P}_4$.

Now consider $J = \{\texttt{a}\}$. The reduct $\mathcal{P}_4^J$ is $\{a \texttt{ v } \neg b \texttt{ :- } c. \;\; ; \;\; a \texttt{ v } c.\}$ and it can be easily verified that J is an answer set of $\mathcal{P}_4^J$, so it is also an answer set of $\mathcal{P}_4$.

If, on the other hand, we take $K = \{\texttt{c}\}$, the reduct $\mathcal{P}_4^K$ is equal to $\mathcal{P}_4^J$, but K is not an answer set of $\mathcal{P}_4^K$ (for the rule $r : a \texttt{ v } \neg b \texttt{ :- } c$. It holds that $B(r) \subseteq K$, but $H(r) \cap K \neq \emptyset$ is violated). Indeed, it can be verified that I and J are the only answer sets of $\mathcal{P}_4$.

Remark 2.3 In some cases it is possible to emulate disjunction by non-stratified normal rules by "shifting" the disjunction to the body [52]. However, this is not possible in general. For example, consider $\mathcal{P}_5 = \{a \, \mathtt{v} \, b.\}$ and $\mathcal{P}'_5 = \{a \,\texttt{:-}\, \texttt{not}\ b. \ ; \ b \,\texttt{:-}\, \texttt{not}\ a.\}$. Both programs have the same answer sets, namely $\{a\}$ and $\{b\}$.

On the other hand, $\mathcal{P}_6 = \{a \, \mathtt{v} \, b. \ ; \ a \,\texttt{:-}\, b. \ ; \ b \,\texttt{:-}\, a.\}$ has $\{a, b\}$ as its single answer set, while $\mathcal{P}'_6 = \{a \,\texttt{:-}\, \texttt{not}\ b. \ ; \ b \,\texttt{:-}\, \texttt{not}\ a. \ ; \ a \,\texttt{:-}\, b. \ ; \ b \,\texttt{:-}\, a. \ \}$ has no answer set at all.

Given a ground program $\mathcal{P}$ with weak constraints $WC(\mathcal{P})$, we are interested in the answer sets of the part without weak constraints (denoted $Rules(\mathcal{P})$) which minimize the sum of weights of the violated constraints in the highest priority level,[2] and among them those which minimize the sum of weights of the violated constraints in the next lower level, etc. This is expressed by an objective function $H^{\mathcal{P}}_A$ for $\mathcal{P}$ and an answer set A ($f_{\mathcal{P}}$ is an auxiliary function mapping leveled weights to weights without levels):

$$\begin{aligned} f_{\mathcal{P}}(1) &= 1 \\ f_{\mathcal{P}}(n) &= f_{\mathcal{P}}(n-1) \cdot |WC(\mathcal{P})| \cdot w^{\mathcal{P}}_{max} + 1, \quad n > 1 \\ H^{\mathcal{P}}_A &= \sum_{i=1}^{l^{\mathcal{P}}_{max}} (f_{\mathcal{P}}(i) \cdot \sum_{N \in N_i^{A,\mathcal{P}}} w_N) \end{aligned}$$

where $w^{\mathcal{P}}_{max}$ and $l^{\mathcal{P}}_{max}$ denote the maximum weight and maximum level of a weak constraint in $\mathcal{P}$, respectively; $N_i^{A,\mathcal{P}}$ denotes the set of weak constraints in level i which are violated by A, and w_N denotes the weight of the weak constraint N. Note that $|WC(\mathcal{P})| \cdot w^{\mathcal{P}}_{max} + 1$ is greater than the sum of all weights in the program, and therefore guaranteed to be greater than any sum of weights of a single level. If weights in level i are multiplied by $f_{\mathcal{P}}(i)$, it is sufficient to calculate the sum of these updated weights, such that the updated weight of a violated constraint of a greater level is always greater than any sum of updated weights of violated constraints of lower levels.

For a ground DLP^w program (with weak constraints) $\mathcal{P}$, A is an *(optimal) answer set* of $\mathcal{P}$ if and only if (1) A is an answer set of $Rules(\mathcal{P})$ and (2) $H^{\mathcal{P}}_A$ is minimal over all the answer sets of $Rules(\mathcal{P})$.

Example 2.4 Consider the following program $\mathcal{P}_{wc}$, which contains three weak constraints.

$a \, \mathtt{v} \, b.$
$b \, \mathtt{v} \, c.$
$d \, \mathtt{v} \, \neg d \ \texttt{:-} \ a, c.$
$:\sim a, c. \ [2:1]$
$:\sim \neg d. \ [1:1]$
$:\sim b. \ [3:1]$

$Rules(\mathcal{P})[wc]$ admits three answer sets: $A_1 = \{a, c, d\}$, $A_2 = \{a, c, \neg d\}$, and $A_3 = \{b\}$. We have: $H^{\mathcal{P}_{wc}}_{A_1} = 2$, $H^{\mathcal{P}_{wc}}_{A_2} = 3$, $H^{\mathcal{P}_{wc}}_{A_3} = 3$.

Thus, the unique (optimal) answer set is $\{a, c, d\}$ with weight 2 in level 1.

[2]Note that higher values for weights and priority levels mark weak constraints of higher importance. E.g., the most important constraints are those having the highest weight among those with the highest priority level.

3 Declarative Programming through Computational Logic

In this section, we review the usage of computational logic as a tool for knowledge representation and reasoning. We first deal with a couple of classical deductive database applications. Then, we present a new programming methodology, which allows us to encode complex queries and, more general, search problems in a simple and highly declarative fashion; even some optimization problems of very high computational complexity (up to Δ_3^P) can be declaratively encoded by using this methodology. Finally, we illustrate the usage of the methodology on a number of complex problems.

3.1 Deductive Database Applications

First, we will present how computational logic might be exploited to solve two problems motivated by classical deductive database applications, namely Reachability and Same Generation. Both can be encoded by just using (positive) datalog rules, thus just scratching to surface of the expressiveness of DLP^w for knowledge representation.

3.1.1 Reachability (REACH)

Given a directed graph $G = (V, E)$ we want to compute all pairs of nodes (a, b) such that b is reachable from a through a sequence of edges in E. In database terms, the problem amounts to computing the transitive closure of the relation containing the edges.

As input we assume a relation $edge(X, Y)$ where a fact $edge(a, b)$ means that there is an edge from a to b in G; the set of vertices is given implicitly by this relation.

The following program then computes a relation $reachable(X, Y)$ containing all facts of the form $reachable(a, b)$ such that b is reachable from a through the edges of the input graph G:

$$\begin{array}{l} reachable(X,Y) \text{ :- } edge(X,Y). \\ reachable(X,Y) \text{ :- } edge(X,U), reachable(U,Y). \end{array}$$

3.1.2 Same Generation (SAMEGEN)

Given a parent-child relationship (an acyclic directed graph), we want to find all pairs of persons belonging to the same generation. Two persons are of the same generation if (i) either they are siblings, or (ii) 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 a parent of $moritz$.

This problem can be encoded by the following program which computes a relation $samegeneration(X, Y)$ containing all facts such that X is of the same generation as Y:

$$\begin{array}{l} samegeneration(X,Y) \text{ :- } parent(P,X), parent(P,Y). \\ samegeneration(X,Y) \text{ :- } parent(P1,X), parent(P2,Y), samegeneration(P1,P2). \end{array}$$

3.2 A Declarative Programming Methodology

Disjunctive datalog programs can be used to encode problems in a highly declarative fashion, following a "Guess/Check/Optimize" paradigm. In this section, we describe this technique, and in the next subsection we illustrate how to apply it on a number of examples. We will see that many problems, also problems of comparatively high computational complexity (that is, even Σ_2^P-complete or Δ_3^P-complete problems), can be solved in a natural manner with DLP^w by using this declarative programming methodology. The power of disjunctive rules allows for expressing problems which are even more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so uniformly over varying instances.

Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem $\mathbf{P}$, a Guess/ Check/ Optimize program $\mathcal{P}$ for $\mathbf{P}$ consists of the following three parts:

Guessing Part The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, in a way such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

Checking Part The checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program tests whether a possibly suboptimal solution is in fact a solution, such that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the solutions for the problem instance I.

Optimization Part The optimization part $\mathcal{O} \subseteq \mathcal{P}$ of the program assigns costs to the individual resources represented in $\mathcal{P}$ (by means of weak constraints), permitting to filter solutions for minimality w.r.t. total cost.

The three layers above can also use additional auxiliary predicates, which can be seen as a background knowledge.

The optimization part, by the definition of the semantics of weak constraints, is clearly an add-on. Let us first focus on the other two parts in more detail.

In general, we may allow both $\mathcal{G}$ and $\mathcal{C}$ to be arbitrary collections of rules (and, for the optimization part, weak constraints) in the program, and it may depend on the complexity of the problem which kinds of rules are needed to realize these parts (in particular, the checking part); we defer this discussion to a later point in this section.

Without imposing restrictions on which rules $\mathcal{G}$ and $\mathcal{C}$ may contain, in the extremal case we might set $\mathcal{G}$ to the full program and let $\mathcal{C}$ be empty, i.e., all checking is integrated into the guessing part such that solution candidates are always solutions. However, in general the generation of the search space may be guarded by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue any further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

For many problems, however, a natural Guess/Check/Optimize program can be designed, in which the three parts are clearly identifiable and have a simple structure:

- The guessing part $\mathcal{G}$ consists of some disjunctive rules which "guess" a solution candidate S.

- The checking part $\mathcal{C}$ consists of integrity constraints which check the admissibility of S, possibly using auxiliary predicates which are defined by normal stratified[3] rules.

- The optimization part $\mathcal{O}$ consists of weak constraints expressing the "desiderata" for the solutions, that is, expressing which properties the solutions *should possibly have*. The set $\mathcal{O}$ of weak constraints implicitly defines an objective function over the answer sets of $\mathcal{C} \cup \mathcal{C} \cup \mathcal{F}_I$. The semantics of $\mathcal{C} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ optimizes this function, providing the answer sets minimizing its value.

The three layers use also additional auxiliary predicates, which are defined by normal stratified rules.

Thus, the disjunctive rule defines the search space in which rule applications are branching points, while the integrity constraints prune illegal branches. (Apart from the point made in Remark 2.3 regarding their difference from rules using unstratified negation in terms of semantics, disjunctive rules also have the advantage of being more compact and usually also more natural.) The weak constraints induce an ordering on the answer sets, allowing the user to specify the best solutions according to an optimization criterion.

3.3 Examples

In this section, we illustrate the declarative programming methodology described in Section 3.2 by showing its application on a number of concrete cases.

3.3.1 Hamiltonian Path

Let us start by a classical NP-complete problem in the field of graph theory, namely *Hamiltonian Path*.

[HAMPATH] *Given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?*

Suppose that the graph G is specified by using predicates *node* (unary) and *arc* (binary), and the starting node is specified by the predicate *start* (unary). Then, the following Guess/Check/Optimize program $\mathcal{P}_{hp}$ solves the problem HAMPATH. [4]

$$inPath(X,Y) \;\mathtt{v}\; outPath(X,Y) \;\text{:-}\; arc(X,Y). \quad \Big\} \textbf{ Guess}$$

$$\left.\begin{array}{l}
\left.\begin{array}{l}
\text{:-}\; inPath(X,Y), inPath(X,Y1), Y <> Y1.\\
\text{:-}\; inPath(X,Y), inPath(X1,Y), X <> X1.\\
\text{:-}\; node(X), \mathtt{not}\; reached(X).
\end{array}\right\} \textbf{Constraints}\\
\left.\begin{array}{l}
reached(X) \;\text{:-}\; start(X).\\
reached(X) \;\text{:-}\; reached(Y), inPath(Y,X).
\end{array}\right\} \begin{array}{l}\textbf{Auxiliary}\\ \textbf{Predicate}\end{array}
\end{array}\right\} \textbf{Check}$$

The first rule guesses a subset S of all given arcs to be in the path, while the rest of the program checks whether that subset S constitutes a Hamiltonian Path. Here, the checking part $\mathcal{C}$ uses an auxiliary predicate *reached*, which is defined using positive recursion.

[3]For a definition of stratification, see [3].
[4]Note that the Optimization part is empty in this program.

In particular, the first two constraints in $\mathcal{C}$ check whether the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian Path must satisfy: There must not be two arcs starting at the same node, and there must not be two arcs ending in the same node.

The two rules after the constraints define the reachability relation from the starting node with respect to the set of arcs S. This relation is used in the third constraint, which enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. This constraint also ensures that this subgraph is connected.

It is easy to see that any set of arcs S which satisfies all three constraints must contain the arcs of a path $p = v_0, v_1, \ldots, v_k$ in G that starts at node a, and passes through distinct nodes until no further node is left, or it arrives at the starting node a again. In the latter case, this means that the path is a Hamiltonian Cycle, and by dropping the last arc, we have a Hamiltonian Path.

Thus, given a set of facts $\mathcal{F}$ for *node*, *arc*, and *start* which specify the problem input, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the input graph has a Hamiltonian Path.

We remark that if we want to compute a Hamiltonian Path rather than only answering whether such a path exists, we can strip off the last arc from a Hamiltonian Cycle by adding a further constraint $\text{:-}\ start(Y), inPath(\_, Y).$ to the program. Then, the set S of selected arcs in an answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at a.

3.3.2 Travelling Salesperson

A DLP^w encoding for the Travelling Salesperson Problem (TSP) can be easily obtained by realizing its definition as the optimization-problem version of HAMPATH (where each edge in the graph carries a weight and the HAMPATH is selected which carries the minimum total weight).

The encoding is obtained by extending the above HAMPATH encoding (containing both guess and check parts) by an optimization part consisting of a single weak constraint

$$:\sim\ inPath(X, Y, C).\ [C : 1]$$

which filters out all answer sets for which the sum of the costs of arcs selected (using inPath) is minimal. Naturally, arc, inPath, and outPath are now ternary, with a third column representing the path costs.

3.3.3 Strategic Companies

The two problems considered so far are, respectively, NP-hard and Δ_2^P-hard, problems. We next show that also harder problems, located at the second level of the polynomial hierarchy, can be encoded by the Guess/Check/Optimize technique. To this end, we consider the following problem *Strategic Companies*.

[STRATCOMP] [14] *Given the collection $C = \{c_1, \ldots, c_m\}$ of companies c_i owned by a holding, the set $G = \{g_1, \ldots, g_n\}$ of goods, and for each c_i a set $G_i \subseteq G$ of goods produced by c_i, and a set $O_i \subseteq C$ of companies controlling (owning) c_i. This control can be thought of as a majority in shares; companies not in C, which we do not model here, might have* *shares in companies as well.* O_i *is referred to as the* controlling set *of* c_i*. Note that, in general, a company might have more than one controlling set, and only irredundant controlling sets (i.e., no proper subset is a controlling set) are recorded then. Let the holding produce all goods in* G*, i.e.* $G = \bigcup_{c_i \in C} G_i$.

A subset of the companies $C' \subset C$ *is a* production-preserving *set if the following conditions hold: (1) The companies in* C' *produce all goods in* G*, i.e.,* $\bigcup_{c_i \in C'} G_i = G$*. (2) The companies in* C' *are closed under the controlling relation, i.e. if* $O_i \subseteq C'$ *for some* $i = 1, \ldots, m$ *then* $c_i \in C'$ *must hold.*

A subset-minimal set C'*, which is* production-preserving*, is called a* strategic set*. A company* $c_i \in C$ *is called* strategic*, if it belongs to some strategic set of* C*.*

Computing the set of all strategic companies is relevant when companies should be sold, as selling any company which is strategic would for sure lead to a violation of any of conditions (1) and (2). This problem is Σ_2^P-hard in general [14]; reformulated as a decision problem ("Given a particular company c in the input, is c strategic?"), it is Σ_2^P-complete. To our knowledge, it is one of the rare KR problems from the business domain of this complexity that have been considered so far.

In the following, we adopt the setting from [14] where each product is produced by at most two companies (for each $g \in G$ $|\{c_i \mid g \in G_i\}| \leq 2$) and each company is jointly controlled by at most three other companies, i.e. $|O_i| \leq 3$ for $i = 1, \ldots, m$ (in this case, the problem is still Σ_2^P-hard). Assume that $\mathcal{F}$ for a given instance of STRATCOMP contains the following facts:

- $company(c)$ for each $c \in C$
- $prod\_by(g, c_j, c_k)$ if $\{c_i \mid g \in G_i\} = \{c_j, c_k\}$ and $c_j \neq c_k$
  $prod\_by(g, c_j, c_j)$ if $\{c_i \mid g \in G_i\} = \{c_j\}$
- $contr\_by(c_i, c_k, c_m, c_n)$ if $c_i \in C$ and $O_i = \{c_k, c_m, c_n\}$ and $c_k \neq c_m \neq c_n$
  $contr\_by(c_i, c_k, c_m, c_m)$ if $c_i \in C$ and $O_i = \{c_k, c_m\}$ and $c_k \neq c_m$
  $contr\_by(c_i, c_k, c_k, c_k)$ if $c_i \in C$ and $O_i = \{c_k\}$

We next present a program, $\mathcal{P}_{strat}$, which solves the complex problem STRATCOMP in a surprisingly elegant way by only two rules:

$$r_{s1}: \quad \left.strat(Y) \vee strat(Z) \mathrel{:-} prod\_by(X, Y, Z).\right\} \textbf{Guess}$$

$$r_{s2}: \quad \left.\begin{array}{l} strat(W) \mathrel{:-} contr\_by(W, X, Y, Z), \\ \qquad\qquad\quad strat(X), strat(Y), strat(Z). \end{array}\right\} \textbf{``Check''}$$

Here $strat(X)$ means that company X is a strategic company. The guessing part $\mathcal{G}$ of the program consists of the disjunctive rule r_{s1}, and the checking part $\mathcal{C}$ consists of the normal rule r_{s2}. The program $\mathcal{P}_{strat}$ exploits the minimization which is inherent to the semantics of answer sets for the check whether a candidate set C' of companies that produces all goods and obeys company control is also minimal with respect to this property.

The guessing rule r_{s1} intuitively selects one of the companies c_1 and c_2 that produce some item g, which is described by $prod\_by(g, c_1, c_2)$. If there were no company control information, minimality of answer sets would then naturally ensure that the answer sets of $\mathcal{F} \cup \{r_{s1}\}$ correspond to the strategic sets; no further checking would be needed. However, in case such control information (given by facts $contr\_by(c, c_1, c_2, c_3)$) is available, the rule r_{s2} in the program checks that no company is sold that would be controlled by other companies in the strategic set, by simply requesting that this company must be strategic as well. The minimality of the strategic sets is automatically ensured by the minimality of answer sets.

Proposition 3.1 *The answer sets of $\mathcal{P}_{strat} \cup \mathcal{F}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}$. Thus, a company c is strategic if $strat(c)$ is a brave consequence, i.e. $\mathcal{P}_{strat} \cup \mathcal{F} \models_b strat(c)$ holds.*

An important note here is that the checking "constraint" r_{s2} interferes with the guessing rule r_{s1}: applying r_{s2} may "spoil" the minimal answer set generated by rule r_{s1}. For example, suppose the guessing part gives rise to a ground rule r_{sg1}

$$strat(c1) \ \texttt{v} \ strat(c2) \mathrel{:-} prod\_by(g, c1, c2).$$

and the fact $prod\_by(g, c1, c2)$ is given in $\mathcal{F}$. Now suppose the rule is satisfied in the guessing part by making $strat(c1)$ true. If, however, in the checking part an instance of rule r_{s2} is applied which derives $strat(c2)$, then the application of the rule r_{sg1} to derive $strat(c1)$ is invalidated, as the minimality of answer sets implies that not both $strat(c1)$ and $strat(c2)$ can be derived from r_{sg1}.

"Feedback" or "influence" of the checking part $\mathcal{C}$ on the guessing part $\mathcal{G}$, in terms of literals which are derived in $\mathcal{C}$ and invalidate the application of rules in $\mathcal{G}$ or make further rules in $\mathcal{G}$ applicable (and thus change the guess), can be made precise in terms of a "potentially uses" relation [26, 28] and a "splitting set" [54]. Such interference is in fact needed to solve STRATCOMP in the way we have outlined, if $\mathcal{C}$ does not contain disjunctive rules. This follows from complexity considerations: If $\mathcal{C}$ had no influence on $\mathcal{G}$ (augmented by $\mathcal{F}$), then the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}$ would just be the answers sets of $\mathcal{C} \cup A_0$, where A_0 is an answer set of $\mathcal{G} \cup \mathcal{F}$. Furthermore, if $\mathcal{C}$ contains no disjunctive rule and $\mathcal{G}$ has complexity in NP (e.g., if $\mathcal{G}$ is, as in this case, head-cycle-free [11]), along with A_0 an answer set A of $\mathcal{C} \cup A_0$ can be guessed and both A_0 and A can be verified in polynomial time. The complexity of deciding whether an atom belongs to some answer set of such a feedback-free program is thus in NP. Now we recall that a particular company is strategic iff it belongs to some strategic set, and deciding this is a Σ_2^P-complete problem. Unless $\Sigma_2^P = \mathrm{NP}$, a hypothesis that is widely believed to be false, we can thus conclude that it is impossible to solve the STRATCOMP problem by a program whose answer sets correspond to the strategic sets, but in which $\mathcal{C}$ has no influence on $\mathcal{G}$.

In general, if a program encodes a problem that is Σ_2^P-complete, then the checking part $\mathcal{C}$ must contain disjunctive rules unless $\mathcal{C}$ has influence on the guessing part $\mathcal{G}$ (or $\mathcal{G}$ has Σ_2^P-complexity). In particular, if the above program $\mathcal{P}_{strat}$ is rewritten to eliminate such influence, then further disjunctive rules must be added. An encoding which expresses the strategic sets in the generic Guess/Check/Optimize-paradigm, where the guessing and the checking part are clearly separated, can be found in [27].

4 Relevant features of computational logic systems

When talking about computational logic, people refer to all uses of logic in computer science. Some of the fields in which computational logic has been actively exploited are artificial intelligence, computational complexity, programming languages, automated software verification, constraint programming, concurrency calculi and tools, model checking, proof theory, information integration and so on.

This wide range of possible applications of computational logic makes it difficult to perform an objective comparison between different systems without fixing the context in which the systems are to be compared.

In this section we single out some of the features that, in our opinion, are relevant to characterize computational logic systems when applied to the context of information integration.

An important aspect of the data integration field consists of answering queries involving several data sources rather than one single database. As an example, the set of informations of interest for a user might be distributed over a network. It is necessary to bring data distributed over a network to a user's machine so that the data may be manipulated to answer user queries. In a distributed environment, it is likely that one will want to save answers to queries in the local machine's cached database for answers, rather than to have to access data over the network to answer the query. In this situation the resources are the cached relations (i.e. views on the source data) and the use of these resources is an important aspect of query answering.

In some data integration systems the database relations are views themselves (either virtual or materialized) on other resources; thus, it is necessary to follow chains of views in order to reach the original source data. In this context, the **expressiveness of the language** supported by the computational logic system plays a central role.

One of the main problems that must be faced in data integration activities concerns the possible presence of data inconsistencies and incompleteness [15, 51, 50]. Indeed, when the data stored in autonomous and heterogeneous sources are considered as a whole and put together in a global scheme they might result inconsistent due to the presence of integrity constraints expressed over the global scheme. Generally speaking, when data are considered locally, they must satisfy only the integrity constraints specified for the source which they belong to, but they are not required to satisfy the integrity constraints expressed in the global scheme. Hence, given a set of data sources as part of an integration system, and a set of global requirements which the integrated data must satisfy, inconsistencies may arise, and these have to be properly taken into account during query processing.

As for the current state of the art in data integration, the problem of dealing with inconsistent and incomplete information in data sources has not been completely approached in a formal way. For instance, in many proposals for data integration it is generally assumed that the mappings between the sources and the global scheme are "exact", i.e., both sound and complete (with respect to the semantics of the source): possible conflicts and inconsistencies among sources are treated at a lower level, in the so-called "data cleaning" phase, which does not take into account the semantics of the whole integration system. Moreover, common approaches to data integration do not take properly into account the role and the interpretation of the mapping between the sources and the global scheme, i.e. they neglect both the form and the semantics of the mapping. In order to define appropriate integration methods dealing with such complex scenarios, it is necessary the availability of *nonmonotonic* features of the formalism used for expressing both data sources and queries. However, current integration systems do not support these features.

This line of reasoning puts into evidence that, due to the presence of inconsistent or incomplete data, the availability of constructs in the query language allowing to express queries whose complexity might go beyond polynomial time complexity (generally NP/CONP queries), is mandatory. Thus, the possibility to express computationally hard queries is a crucial property that computational logic systems have to provide in order to deal with such complex scenarios.

In general, answering queries in data integration systems is a complex task. Difficulties arise either because of the huge amounts of data to analyze or because the queries to analyze are complex. The **efficiency of the computational logic system** in answering queries is an important measure for characterizing it in data integration activities.

Thus, the systems' features to be considered to evaluate the suitability of a computational logic system for data integration tasks fall in two classes: *(i)* language expressiveness and *(ii)* efficiency. In the sequel of this section we specify more precisely the features to be considered in each of the above classes.

4.1 Language expressiveness

In order to characterize the expressiveness of logic languages in data integration contexts we consider the following properties that a logic language should have to provide the tools for properly solve usual data integration problems:

- *Ability to express views.* As we have outlined above, views play a relevant role in data integration activities; indeed the integrated scheme and/or the source schemes might be represented as views.

- *Ability to express recursive queries.* Since queries are usually defined on views and since views can be obtained from other (either virtual or materialized) views, it is necessary to be able to specify recursive queries on views in order to retrieve all the interesting information.

- *Ability to express integrity constraints.* When integrating data from different sources, differences and inconsistencies of the data present in the available resources must be taken into account. One way to guarantee the consistency of the integrated data and to guide the integration task is to impose integrity constraints on the scheme describing the integrated database. The ability to express integrity constraints makes the system powerful enough to avoid some classes of data inconsistencies.

- *Ability to express nonmonotonic queries.* This property is necessary in order to define appropriate methods dealing with complex scenarios involving incomplete or inconsistent data sources to be integrated.

Language expressiveness	
Ability to express views	
Ability to express recursive queries	
Ability to express integrity constraints	
Ability to express nonmonotonic queries	
Ability to deal with computationally hard queries	
Other	
Efficiency	
Efficiency with simple but data intensive queries	
Efficiency with computationally hard queries	

Table 1: Template table for describing systems' features

- *Ability to deal with computationally hard queries.* In order to handle inconsistencies of data present in the resources, it is necessary to allow users of being aware of such inconsistencies and to provide them with tools for the manipulation of such inconsistent data. In this context, simple queries (i.e., queries of polynomial time complexity) do not serve the purpose. Indeed, the availability of constructs allowing to express queries having complexity in NP/CONP or higher is necessary .

4.2 Efficiency issues

The efficiency of a computational logic system applied to data integration can be measured by analyzing its behaviour in query answering. In this setting, there are basically two parameters which efficiency depends on: *(i)* the quantity of data to be analyzed for answering the query and *(ii)* the intrinsic complexity of the query. In this report, in order to analyze the efficiency of the considered computational logic systems, we consider their behaviour in answering two particular classes of queries:

- simple queries (i.e., queries that can be answered in polynomial time) over large amounts of data;
- complex queries (i.e., whose complexity goes beyond polynomial-time) over medium-sized data.

4.3 Summary of the systems' features to be evaluated

In the sequel of the report we describe some of the most important computational logic systems proposed in the literature and, for each of them, we analyze the features outlined in the previous sections. In particular, we summarize these features in a table as shown in Table 1 and describe, for each of them, the characteristics of the system.

5 Overview of declarative computational logic systems

5.1 Deductive database systems

Database and logic programming are two independently developed areas in computer science. Database technology has evolved in order to effectively, efficiently and reliably organize, manage and maintain large volumes of increasingly complex data in various memory devices. Logic programming is a direct outgrowth of earlier work in automatic theorem proving and artificial intelligence. It is based on mathematical logic, which is the study of the relations between assertions and deductions and is formalized in terms of proof and model theories.

Important studies on the relations between logic programming and relational databases have been conducted for about two decades, mostly from a theoretical point of view [34, 33, 43, 87, 63]. Relational databases and logic programming have been found quite similar in their representation of data at the language level. They have also been found complementary in many aspects. Relational database systems are superior to the standard implementations of logic programming systems with respect to data independence, secondary storage access, concurrency, recovery, security and integrity [86]. The control over the execution of query languages is up to the system, which uses query optimization and compilation techniques to ensure efficient performance over a wide range of storage structures. However, the expressive power and functionality of relational database query languages are limited when compared to logic programming languages. Relational languages do not have built-in reasoning capabilities. Moreover, relational query languages are often powerless to express complete applications, and are thus embedded in traditional programming languages.

The integration of logic programming and relational database techniques has led to the active research area of *deductive databases* [35, 16, 40]. This combines the benefits of the two approaches such as representational and operational uniformity, reasoning capabilities, recursion, declarative querying, efficient secondary storage access, etc.

A restricted form of Prolog without function symbols called Datalog (with negation), has been widely accepted as deductive database language. It has a well-defined declarative semantic based on the work in logic programming [89, 16].

In the past few years, various set-oriented evaluation strategies specific for deductive databases have been the main focus of extensive research [6, 7, 10, 16, 42, 45, 67, 80, 89, 88] and a number of deductive database systems or prototypes based on Datalog have been reported. These include Nail [66], LOLA [32], Glue-Nail [23], XSB [82], CORAL [75], Aditi [91], LogicBase [41], Declare/SDS [47], EKS(-V1) [96] etc. See [76] for a survey of these deductive database systems.

Many organizations spent some efforts in the field of deductive database systems. As an example, efforts at ECRC led to the study of query evaluation methods (QSD/LSD and others) [92, 93, 94, 49], integrity checking (Soundcheck) [21], the deductive database system EKS(-V1) [96], hypothetical reasoning and integrity constraints checking [95]. The EKS system used a top-down evaluation method.

Efforts at MCC emphasized bottom-up evaluation methods [86] and query evaluation using methods such as seminaive evaluation, magic sets and counting [6, 10, 79, 81], semantics for stratified negation and set-grouping [9], investigation of safety, the finiteness of answer sets and joint order optimization. These studies led to LDL, the first operational deductive database system that was widely available.

Implementation at Stanford started on a system called NAIL! (Not Another Implementation of Logic) [23]. The results of this work led to the first paper on recursion using the Magic Sets method [6], done in collaboration with the group at MCC. Other contributions were aggregation in logical rules, and theoretical contributions to negation: stratified negation [36], well founded negation [37] and modularly stratified negation [78]. A language, called GLUE [72, 65] was developed. GLUE is a language for logical rules that has the power of SQL statements, together with conventional language that permits the construction of loops, procedures and modules.

The University of Wisconsin spent its efforts to the development of CORAL [75]. Bottom-up and magic set methods were implemented. The system is extensible and provides aggregation modularly stratified databases. CORAL supports a declarative language and an interface to C++ which allows for a combination of declarative and imperative programming. The declarative query language supports general Horn clauses augmented with complex terms, set grouping, aggregation, negation and relations with tuples that contain universally quantified variables. CORAL supports a wide range of evaluation strategies and automatically chooses an efficient evaluation strategy.

Deductive database systems have been used in a variety of application domains including scientific modeling, financial analysis, decision support, language analysis, parsing and various applications of transitive closure such as bill-of-materials and path problems [74]. They provide declarativeness and ability to express both views and recursive queries. However they have, generally, limited capabilities to express integrity constraints and are not able to deal with computationally hard queries. As far as efficiency is concerned, they are able to efficiently answer simple but data intensive queries; efficiency on recursive queries is achieved through sophisticated rewriting techniques based on Magic Sets [6, 10].

In the next subsections we give an overview of the main deductive database systems.

5.1.1 LDL

The LDL project [18] is directed towards two significant goals. The first one is the design of Logical Data Language (LDL), a declarative language for data-intensive applications which extends pure Datalog with sets, negation and updates. The second goal is the development of a system supporting LDL, which integrates rule base programming with efficient secondary memory access, transaction management recovery and integrity control. The LDL system belongs properly to the class of integrated systems; the underlying database engine is based on relational algebra and was developed specifically within the LDL project.

The LDL language supports complex terms within facts and rules and stratified negation. Programs which incorporate negation but are not stratified are regarded as inadmissible programs. Moreover, LDL supports updates through special rules. The body of an update rule incorporates two parts, a *query part* and a *procedure part*. The former part identifies the tuples to be updated, and should executed first; as an effect of the query part, all variables of the procedure part should get bound. Then, the procedure part is applied to the underlying database; the procedure part contains *insert* operations and *delete* operations. The order of execution of the procedure is left-to-right, unless the system detects that the procedure has the *Church-Rosser* property (namely the procedure produces the same answer for every permutation of its components); in this case, the most efficient order of execution is decided by the system.

The LDL compiler uses the notion of *dynamic adornments*, by knowing which places will be bound to constant values at goal execution; these places will have a *deferred constant*. This is quite similar to the optimization of parametric queries which takes place in conventional database systems. With nonrecursive rules the main problem considered by the optimizer is determining the order and method for each join. With recursive rules, the compiler tries to push selection constants. This entails searching for recursive rules equivalent to the original ones; the search considers some special, simple changes of the rule structure, thus producing programs which are semantically equivalent to the given one. If these methods fail, then the LDL optimizer uses methods of *magic sets* and *counting* for rule rewriting.

Special attention is dedicated to rules containing set terms. Each rule containing a set term is replaced at compile time by a set of rules, each containing ordinary terms; in this way, matching is reduced to ordinary matching. However, several ordinary matching patterns are required for dealing with each set matching. In order to reduce them, elements of sets within facts are stored in ascending ascii order.

The improved LDL program, produced after the application of various rewriting methods, is translated into an expression of relational algebra extended by the fixpoint operation; this expression is given as input to the optimizer. Relational expressions are internally represented through *processing graphs* whose nodes correspond either to database relations or to operations; edges indicate relationships between operations and their operands. A particular transformation, called *contraction*, turns processing graphs into trees, by substituting cycles with special nodes corresponding to fixpoints.

All selections and projections that are directly applicable to database relations are included in the final processing strategy and not further considered in the optimization. In spite of this simplifications, there is an extremely high number of *processing trees* equivalent to the given one, obtained by permuting the order of evaluation of joins and fixpoints, and by selecting the appropriate technique for computing joins and fixpoints among various available methods.

The execution environment for LDL is an algebraic machine, which is capable of performing retrieval and manipulation of complex terms, as well as efficient joins and unions.

In Table 2 we summarize the features of the LDL system w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.1.2 NAIL!

The NAIL! (Not Another Implementation of Logic) [23] project at Stanford University aims at supporting the optimal execution of Datalog goals over a relational database sys-

LDL	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	To some extent
Ability to express nonmonotonic queries	Only stratified negation
Ability to deal with computationally hard queries	No
Other	Supports set terms and unification
Efficiency	
Efficiency with simple but data intensive queries	Yes - Magic sets techniques are also supported
Efficiency with computationally hard queries	Not applicable

Table 2: Features of LDL

tem. The fundamental assumption of the project is that *no single strategy is appropriate for all logic programs*, hence, much attention is devoted to the development of an *extensible* architecture, which can be enhanced through progressive additions of novel strategies.

The language supported by NAIL! is stratified Datalog extended by function symbols, negation (through the *not* operator) and sets (through the *findall* operator); rules must be satisfied with respect to the *not* and *findall* operator.

NAIL! does not support set terms or set unification, as LDL does; the *findall* predicate really produces a list, though the language does not control the order of elements within the list. The *findall* operator can be used to produce the same effect as the grouping in SQL.

The NAIL! system considers a Datalog program and a user's goal as input. Each predicate of each group of mutually recursive predicates is considered separately. From the user's goal, the system generates the *adorned goal*, a synthetic representation consisting of the name of the goal predicate with a superscript string; letters b and f in the superscript denote places which are respectively *bound* and *free*. It should be noted that the computation of an adorned goal p might require the computation of other adorned goals over other predicates q, or even of different adornments for p.

The NAIL! system uses *capture rules* as basic strategies. Each capture rule applies to particular *adorned goals*, and produces the corresponding result relations. The evaluation of each capture rule involves a pair of algorithms; the first one concerns the *applicability* of a capture rule, the second one concerns the *substantiation* of the capture rule, namely the evaluation of the result relation. A capture rule is applicable to an adorned goal if the rule can evaluate the result relation with a finite number of iterations; thus, applicability algorithms express sufficient conditions for convergence of substantiation algorithms. Applicability is based on the adorned goal only, while substantiation requires the notion of the particular goal constraint.

There are basically four kinds of substantiation rules:

- *basic* rules, to be used to "capture" nodes which represent nonrecursive predicates;

NAIL!	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	To some extent
Ability to express nonmonotonic queries	Only stratified negation
Ability to deal with computationally hard queries	No
Other	Aggregation in logical rules, supports sets
Efficiency	
Efficiency with simple but data intensive queries	Yes - Magic sets techniques are also supported
Efficiency with computationally hard queries	Not applicable

Table 3: Features of NAIL!

- *sideways* rules, which use *sideways information passing* to transmit bindings among goals;
- *top-down* rules, which use resolution;
- *bottom-up* rules, which use forward chaining.

This classification includes a variety of evaluation methods; capture rules can be considered a flexible and extensible *control mechanism* for the execution of logic goals.

In Table 3 we summarize the features of the NAIL! system w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.1.3 POSTGRES

POSTGRES [73] is a large project for developing a new generation database system, extending relational technology to support complex objects, data types, rules, versions, historic data, and new storage media.

POSTGRES is the successor of the INGRES project and, in particular, it provides extensions to the QUEL query language of INGRES. New POSTQUEL commands are used for specifying iterative execution of queries, alerts, triggers and rules. "Iterative" queries support transitive closures; linguistically, this is achieved by appending an asterisk to each command that should be iteratively executed. Alerts and triggers are commands which are active indefinitely and may be "awakened" as a consequence of database changes. The system supports their efficient monitoring.

The implementation of alerts and triggers is done through special locks, which are placed on the data. Whenever a transaction writes a data object which is locked, the corresponding "dormant" commands are activated. The lock manager maintains information about dormant commands stored within the tables of the system.

POSTGRES	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	In the form supported by SQL99
Ability to express integrity constraints	To some extent, SQL99 constraints are supported
Ability to express nonmonotonic queries	No
Ability to deal with computationally hard queries	No
Other	Supports Triggers and Alerts
Efficiency	
Efficiency with simple but data intensive queries	Yes
Efficiency with computationally hard queries	Not applicable

Table 4: Features of POSTGRES

In Table 4 we summarize the features of POSTGRES w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.2 Answer set programming systems

Answer set programming (ASP) is a declarative approach to programming, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. Answer set programming[5] (ASP) is a logic programming alternative to SAT-based programming, which is successful and widely used in the area of Artificial Intelligence [46]. In SAT-based programming, a given computational problem P is encoded as a propositional CNF formula whose models correspond to solutions of P; a SAT solver is then used to find such models (and thus solution of P). In answer set programming, instead, a given computational problem P is represented by an ASP program whose answer sets[6] correspond to solutions; an ASP system is then used to find such solutions of P [55].

The main advantage of answer set programming over SAT-based programming is the higher expressiveness of its language, which enjoys the knowledge modelling power of logic programming features like variables, negation as failure, and disjunction. Indeed, the knowledge representation language of ASP consists of function-free logic programs with classical negation where disjunction is allowed in the heads and negation as failure may occur in the bodies of the rules.

The ASP language supports the representation of problems of high computational complexity (specifically, all problems in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$ [28]). Importantly, the ASP encoding of a large variety of problems is often very concise, simple, and elegant [27].

The most widely used ASP systems are DLV and SMODELS. In the sequel of this section we provide an overview of ASP systems focusing more in depth on the most

[5]Name coined by Vladimir Lifschitz in the invited talk at ICLP'99.
[6]An answer set is a preferred model of an ASP program.

popular systems **DLV** and SMODELS.

5.2.1 DLV

The development of the **DLV** system (datalog plus vel, i.e., disjunction) [12, 20, 22] has started as a research project financed by FWF (the Austrian Science Funds) in 1996, and has evolved into an international collaboration over the years. Currently, the University of Calabria and TU Wien participate in the project, supported by a scientific-technological collaboration between Italy and Austria.

The system is based on disjunctive logic programming without function symbols under the consistent answer set semantics [38] and has the following important features:

Advanced Knowledge Modeling Capabilities. **DLV** provides support for declarative problem solving in several respects:

- High expressiveness in a formally precise sense (Σ_2^P), so any such problem can be uniformly solved by a fixed program over varying input.
- Declarative problem solving following a "Guess/Check/Optimize" paradigm where a solution to a problem is guessed by one part of a program and then verified through another part of the program.
- Capability to express hard and weak constraints.
- A number of front-ends for dealing with specific AI applications.

Solid Implementation. Much effort has been spent on sophisticated algorithms and techniques for improving the performance, including

- database optimization techniques [22, 29], and
- non-monotonic reasoning optimization techniques.

Database Interfaces. The **DLV** system provides an experimental interface to an object-oriented database management system (Objectivity), by means of a special query tool, which is useful for the integration of specific problem solvers developed in **DLV** into more complex systems.

The architecture of **DLV** is illustrated in Figure 1. The general flow in this picture is top-down. The principal User Interface is command-line oriented, but also a Graphical User Interface (GUI) for the core systems and most front-ends is available. Subsequently, front-end transformations might be performed. Input data can be supplied by regular files, and also by Objectivity databases. The **DLV** kernel (the shaded part in the figure) then produces answer sets one at a time, and each time an answer set is found, "Filtering" is invoked, which performs post-processing (dependent on the active front-ends) and controls continuation or abortion of the computation.

The **DLV** kernel consists of three major components: The "Intelligent Grounding," "Model Generator," and "Model Checker" modules share a principal data structure, the "Ground Program". It is created by the Intelligent Grounding using differential (and

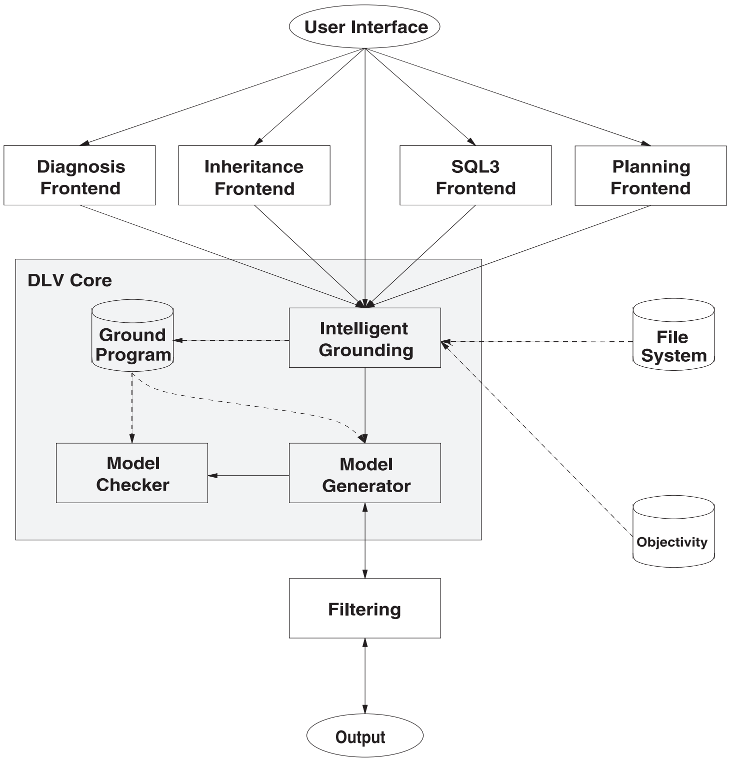


Figure 1: The System Architecture of **DLV**

other advanced) database techniques together with suitable 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) [30, 31], the stability of which are subsequently verified by the latter.

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

In Table 5 we summarize the features of the DLV system w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.2.2 SMODELS

The SMODELS system [70, 69] 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.

As input, the SMODELS system takes logic program rules basically in Prolog style

DLV	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes, both hard and weak constraints (desiderata) are expressible
Ability to express nonmonotonic queries	Yes, supports unstratified negation and disjunction
Ability to deal with computationally hard queries	Expresses very hard queries, up to Δ_3^P
Other	
Efficiency	
Efficiency with simple but data intensive queries	Some database optimization techniques are incorporated, but magic sets are not implemented
Efficiency with computationally hard queries	Heuristics and optimization techniques from the field of SAT programming have been implemented

Table 5: Features of DLV

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 non-recursively.

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 which appears positively in the rule body.

In addition to normal logic program rules, SMODELS supports rules with cardinality and weight constraints. The idea is that, e.g., a cardinality constraint

1 {a,b,not c} 2

holds in an answer set if at least 1 but at most 2 of the literals in the constraint are satisfied in the model and a weight constraint

10 [a=10,b=10,not c=10] 20

holds if the sum of weights of the literals satisfied in the model is between 10 and 20 (inclusive).With built-in functions for integer arithmetic (included in the system), these kind of rules allow compact and fairly straightforward encodings of many interesting problems.

Answer sets of a domain-restricted logic program with variables are computed in three stages:

- First the program is transformed into a ground program without variables.
- Second, the rules of the ground program are translated into primitive rules,
- Third, an answer set is computed using a Davis-Putnam like procedure [84].

The first two stages have been implemented in a program called `lparse`, which functions as a front end to `smodels` which in turn implements the third stage.

In the first stage `lparse` automatically determines the domain predicates and then using database techniques evaluates the domain predicates and creates a ground program which has exactly the same answer sets as the original program with variables. Then the rules are compiled into primitive rules.

The `smodels` procedure is a Davis-Putnam like backtracking search procedure that finds the answer sets of a set of primitive rules by assigning truth values to the atoms of the program. Moreover it uses the properties of the answer set semantics to infer and propagate additional truth values. Since the procedure is in effect traversing a binary search tree, the number of nodes in the search space is in the worst case on the order of 2^n, where n can be taken from the number of atoms that appear in a constraint in a head of a rule or that appear as a negative literal in a recursive loop of the program.

Hence, in order to compute answer sets, one uses the two programs `lparse`, which translates logic programs into an internal format, and `smodels`, which computes the models.

SMODELS	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes, supports unstratified negation
Ability to deal with computationally hard queries	Up to NP thanks to unstratified negation and "choice" rules Σ_2^P problems can be solved through a disjunctive extension (GnT) implemented by a rewriting technique
Other	Supports cardinality and weight constraints; handles only domain-restricted programs
Efficiency	
Efficiency with simple but data intensive queries	Inefficiency caused by the domain-restriction requirement on program variables. No database optimization technique has been implemented
Efficiency with computationally hard queries	Heuristics and optimization techniques from the field of SAT programming have been implemented

Table 6: Features of SMODELS

It is worthwhile noting that, even if SMODELS kernel does not support disjunction, this feature has been implemented on top of SMODELS by a suitable rewriting technique. The resulting system is called GnT [44]. Such a rewriting-based implementation of disjunction, however, can obviously not provide the same performance than a built-in implementation. And indeed, GnT turns out to be sensibly slower than **DLV** on Σ_2^P-hard problems.

In Table 6 we summarize the features of SMODELS w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.2.3 ASSAT

ASSAT (Answer Sets by SAT solvers) [56] is a recently developed system for computing answer sets of a logic program by using SAT solvers. Given a ground logic program P and a SAT solver X, ASSAT(X) works as follows:

- Computes the completion of P and converts it into a set C of clauses.

ASSAT	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes, supports unstratified negation
Ability to deal with computationally hard queries	Up to NP thanks to unstratified negation
Other	
Efficiency	
Efficiency with simple but data intensive queries	No database optimization technique has been implemented
Efficiency with computationally hard queries	By the use of well-assessed SAT solvers

Table 7: Features of ASSAT

- Repeats the following steps
  - Calls X on C to get a model M (terminates with failure if no such M exists).
  - Returns M if M is an answer set.
  - Otherwise, finds some loops in P whose loop formulas are not satisfied by M and adds their corresponding clauses to C.

As shown in [56], this procedure is sound and complete, assuming that X is a sound and complete SAT solver.

ASSAT exploits `lparse`, the grounding system of SMODELS, to instantiate a given program. Then, for each loop in the program which is found during the computation, a corresponding loop formula is added to the program's completion. In this way, a one-to-one correspondence between the answer sets of the program and the models of the resulting propositional theory is obtained. In the worst case, this process requires computing an exponential number of loop formulas.

In Table 7 we summarize the features of ASSAT w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.2.4 noMoRe

The **no**n-**mo**notonic **re**asoning system `noMoRe` [2] implements answer set semantics for normal logic programs. It realizes a novel, rule-based paradigm to obtain answer sets by computing non-standard graph colorings of the *block graph* associated with a given logic program (see [57, 58] for details). These non-standard graph colorings are called *a-colorings* or *application-colorings* since they reflect the set of generating rules (applied rules) for an answer set. Hence `noMoRe` is rule-based and not atom-based like most of the other known systems. It handles *backward propagation* of partial a-colorings and exploit a technique called *jumping* in order to ensure full (backward) propagation [58]. Both techniques improve the search space pruning of `noMoRe`.

The `noMoRe`-system is implemented in the programming language Prolog; it has been developed under the `ECLiPSe` Constraint Logic Programming System [1] and it was also successfully tested with `SWI`-Prolog [98].

`NoMoRe` uses a compilation technique to compute answer sets of a logic program P in three steps. At first, the block graph Γ_P is computed. Secondly, Γ_P is compiled into Prolog code in order to obtain an efficient coloring procedure. Users may choose between two different kinds of compilation, one which is fast but which gives a lot of compiled code and another one which is a little bit slower but which produces less compiled code than the other. The second way of compiling has to be used with large logic programs, depending on the memory management of the underlying Prolog system. The compiled Prolog code (together with the example-independent code) is then used to actually compute the answer sets. To read logic programs it is used a parser (eventually after running a grounder, e.g. `lparse` or `dlv`) and there is a separate part for interpretation of a-colorings into answer sets. Additionally, `noMoRe` comes with an interface to the graph drawing tool `DaVinci` [68] for visualization of block graphs. This allows for a structural analysis of programs.

The `noMoRe` system is used for purposes of research on the underlying paradigm. But even in this early state, usability for anybody familiar with the logic programming paradigm is given. The syntax accepted by `noMoRe` is Prolog-like, like that of **DLV** and `smodels`. Furthermore, `noMoRe` is able to deal with integrity constraints as well as weight and cardinality constraints.

In Table 8 we summarize the features of `noMoRe` w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.2.5 SLG

The SLG system [17] is a research-oriented system for deductive databases and nonmonotonic reasoning. It is built as a meta interpreter on top of existing Prolog systems. In addition to all the functionalities of Prolog, SLG contains several features not usually found in logic programming systems, including:

- Query evaluation under the well-founded semantics using SLG resolution.
- Query evaluation of deductive databases whose rules may have explicit universal quantifiers in the body.
- Query answering under stable models.
- Abductive reasoning with integrity constraints.
- Skeptical reasoning with respect to the intersection of stable models.

The SLG meta interpreter employs an efficient algorithm for incremental maintenance of dependencies among subgoals so that subgoals that are completely evaluated or are possibly involved in recursion through negation can be detected by inspecting the dependency information of a single subgoal.

`noMoRe`	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Unstratified negation is supported
Ability to deal with computationally hard queries	Up to NP through unstratified negation
Other	
Efficiency	
Efficiency with simple but data intensive queries	No database technique has been implemented
Efficiency with computationally hard queries	It is a research prototype implemented in Prolog, which needs some engineering to become efficient

Table 8: Features of `noMoRe`

The SLG-∀ meta interpreter augments the SLG meta interpreter with the handling of universal rules. Traditionally universal quantification is eliminated by conversion into the negation of an existential quantification. This, however, may introduce extra recursion through negation, and does not preserve the alternating fixpoint semantics of general logic programs. SLG-∀ computes the alternating fixpoint semantics by processing universal rules directly.

A profiling of the SLG-∀ meta interpreter shows that two major factors of overhead are meta interpretation and the lack of destructive assignment for managing tables of subgoals and their answers.

In Table 9 we summarize the features of SLG w.r.t. the properties relevant to data integration we have pointed out in Section 4.

5.2.6 DeReS

The system DeReS [19] supports basic automated reasoning tasks for default logic and for logic programming under the answer set semantics. It is shown that a normal logic program P can be represented by a suitable default theory D, such that the answer sets of P correspond to the so-called *extensions* of D. DeReS uses *relaxed stratification* as a primary mechanism for pruning the search-space. A default theory D is partitioned into several smaller subtheories, called *strata* and the extensions of D are constructed from the extensions of its strata. The approach taken by DeReS is somehow orthogonal to the one taken by SMODELS, and it is argued in [19] that next generation implementations of nonmonotonic systems must combine techniques developed in both projects in order to be effective in a large range of different applications.

SLG	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes
Ability to deal with computationally hard queries	Up to NP
Other	
Efficiency	
Efficiency with simple but data intensive queries	No
Efficiency with computationally hard queries	It is a research prototype which needs some engineering to become efficient

Table 9: Features of SLG

The features of the system DeReS are summarized in Table 10.

5.2.7 XSB

The system XSB [77] can compute most cases of the well-founded semantics for normal logic programs with functions symbols. The inference engine, which is called the 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.

Table 11 summarizes the features of XSB relevant to data integration tasks.

5.2.8 Other systems

Among other ASP systems we cite the system *near Horn* [59, 62] which has been implemented in PROLOG; in [83] the system DisLog is described, which incorporates different disjunctive theories and strategies including the semantics introduced in [60]; DisLog tries to eliminate redundant computations by using a breadth-first approach. The system DisLoP [4, 5] which aims at extending the *restart model elimination* and *hyper tableau calculi*, for disjunctive logic programming under the D-WFS and stable semantics. The *aspps* system is an answer-set programming system based on the extended logic of propositional schemes [24], which allows variables but not function symbols in the language.

6 Experiments and benchmarks

As pointed out by the analysis above, deductive database systems are able to deal with larger amounts of data compared to ASP systems but they provide limited capability to

DeReS	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Unstratified negation is supported
Ability to deal with computationally hard queries	Up to NP in Datalog, up to Σ_2^P in Default Logic
Other	Also Default Logic is supported
Efficiency	
Efficiency with simple but data intensive queries	No database optimization technique is implemented
Efficiency with computationally hard queries	An engineering phase is needed

Table 10: Features of DeReS

XSB	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	No
Ability to express nonmonotonic queries	Only "well-founded" queries
Ability to deal with computationally hard queries	Only polynomial-time queries
Other	
Efficiency	
Efficiency with simple but data intensive queries	The system is at an advanced engineering state. However, the "tuple-oriented", top-down computational model limits the efficiency on database queries
Efficiency with computationally hard queries	Not applicable

Table 11: Features of XSB

manage integrity constraints and are not able to answer computationally hard queries as well as to handle inconsistent and incomplete data. On the contrary, the high expressiveness of ASP systems allows them to deal with these problems. Thus, ASP systems seem to be better suited to be exploited in data integration frameworks.

It is widely acknowledged that the most relevant ASP systems, which are nowadays available, are **DLV**, SMODELS and ASSAT.

To evaluate the performance of these systems, we performed a number of benchmarks using the examples provided in Section 3 as well as a couple of further benchmark problems with suitable sets of benchmark data. Wherever possible, we have used the same encoding for all systems, and made only minimal changes to allow for differences among the languages **DLV**, SMODELS, and ASSAT support.

In particular, we have considered the following benchmark problems:

- *Reachability (REACH)*
- *Same Generation (SAMEGEN)*
- *Hamiltonian Path (HAMPATH)*
- *Ramsey Numbers*
- *Strategic Companies (STRATCOMP)*
- *QBF*

The former (REACH and SAMEGEN) are deductive database applications, which should assess the suitability of the systems for simple applications over larger data-instances. The latter are hard problems (NP-complete and Σ_2^P-complete), and are aimed to assess the suitability of the systems to solve the complex tasks arising in data integration.

6.1 Benchmark Data and Encodings

Reachability (REACH) The input graphs for Reachability were generated using the function random_graph(#nodes,#edges,0,0,0,0,0,0,0,0) from the Stanford GraphBase [48], with a ratio of 3:1 between #edges and #nodes.

Same Generation (SAMEGEN) For this problem, we used the encoding described in Section 3 and instances where the parent relation is as depicted in Figure 2.

For SMODELS we had to rewrite the program to account for the limitations of `lparse`:

$$reachable(X,Y) \text{ :- } edge(X,Y).$$
$$reachable(X,Y) \text{ :- } edge(X,U), reachable(U,Y), edge(Dummy,Y).$$

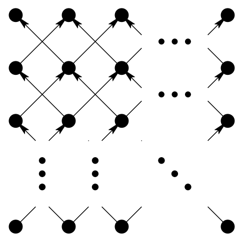


Figure 2: Input for SAMEGEN

Hamiltonian Path (HAMPATH) For DLV, we used the encoding described in Section 3.3.1, while for SMODELS we rewrote the disjunctive guessing rule to use non-stratified negation instead and added several instances of $vertex(\dots)$ to work around the limitations of `lparse`:

$$inPath(X,Y) \text{ :- } \texttt{not } outPath(X,Y), reached(X), arc(X,Y).$$
$$outPath(X,Y) \text{ :- } \texttt{not } inPath(X,Y), reached(X), arc(X,Y).$$

$$\text{:- } inPath(X,Y), inPath(X,Y1), Y <> Y1, vertex(X), vertex(Y), vertex(Y1).$$
$$\text{:- } inPath(X,Y), inPath(X1,Y), X <> X1, vertex(X), vertex(Y), vertex(X1).$$

$$\text{:- } vertex(X), \texttt{not } reached(X).$$
$$reached(X) \text{ :- } start(X).$$
$$reached(X) \text{ :- } inPath(Dummy,X), vertex(Dummy), vertex(X).$$

The graph instances were generated using a tool by Patrik Simons[7] which originally was used to compare SMODELS against SAT solvers [85], and we assumed $vertex(0)$ as the starting node.

Ramsey Numbers The Ramsey number $R(k,m)$ is the least integer n such that, no matter how we color the edges of the complete graph (clique) with n vertices using two colors red and blue, there is a red clique with k vertices (a red k-clique) or a blue clique with m vertices (a blue m-clique). Ramsey numbers exist for all pairs of positive integers k and m.

In our experiments, we considered the problem of deciding, for varying k, m, and n, if n is the ramsey number $R(k,m)$. For example, for **DLV**, we used the following encoding for $R(3,4)$:

$$\begin{array}{rcl} blue(X,Y) \,\texttt{v}\, red(X,Y) & \text{:-} & arc(X,Y). \\ & \text{:-} & red(X,Y), red(Y,Z), red(X,Z). \\ & \text{:-} & blue(W,X), blue(W,Y), blue(W,Z), \\ & & blue(X,Y), blue(Y,Z), blue(X,Z). \end{array}$$

[7] `http://tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`.

Intuitively, the disjunctive rule guesses a color for each arc. The first constraint eliminates the colorings containing a red complete graph with 3 nodes and the second constraint eliminates the colorings containing a blue complete graph with 4 nodes. To check whether an integer n is the ramsey number R(k,m), we add facts $node(1) \cdots node(n)$ to the above program. The program has a stable model if and only if there is a coloring of the edges of the complete graph with n vertices without a monochromatic complete graph with 3 or 4 vertices. So, if there is a stable model for a particular n, then $n < R(3,4)$. The smallest n such that no stable models are found is the Ramsey number $R(3,4)$.

It is worthwhile noting that for this problem we could not use a single, uniform encoding to solve all instances, rather, we have one program for each instance. In particular, for checking that n is the ramsey number $R(k,m)$, the first constraint contains $\binom{k}{2}$ atoms with predicate *red* and the second constraint contains $\binom{m}{2}$ atoms with predicate *blue* .

For SMODELS we replaced the disjunctive rule by two normal rules using non-stratified negation. We also added some instances of $node(\dots)$ similarly as we did for the Hamiltonian Path problem.

Strategic Companies (STRATCOMP) Here, we generated tests with instances for n companies ($5 \leq n \leq 170$), $3n$ products, 10 uniform randomly chosen *controlled_by* relations per company, and uniform randomly chosen *produced_by* relations. To make the problem harder, we only consider strategic sets containing two fixed companies (1 and 2, without loss of generality), enhancing the program $\mathcal{P}_{strat}$ from Section 3.3.3 by the following two constraints:

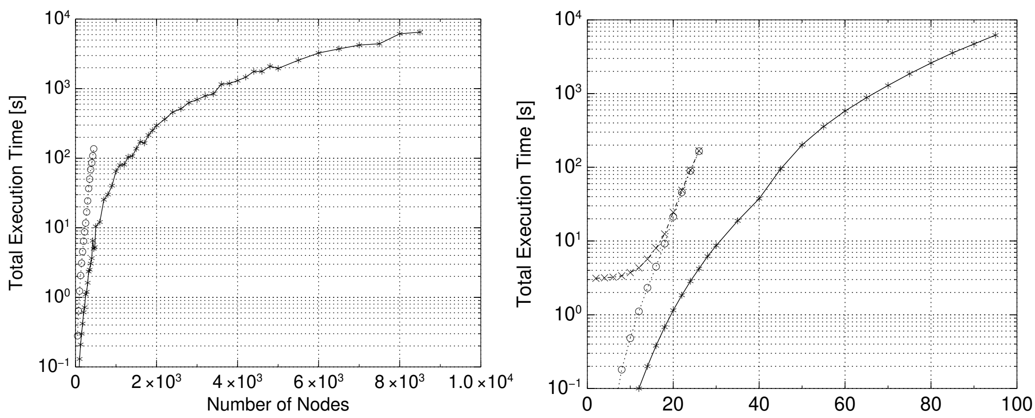
$$:\!-\ \texttt{not}\ strategic(1).$$
$$:\!-\ \texttt{not}\ strategic(2).$$

QBF This problem is well known to be Σ_2^P-complete [71]. The problem here is to decide whether a quantified Boolean formula (QBF) $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a 3DNF formula over $X \cup Y$, is valid.

The transformation from QBF to disjunctive logic programming is a variant of a reduction used in [25] where we clearly separate problem instances and the following general encoding $\mathcal{P}_{QBF}$

$$\begin{array}{rcl}
t(X)\ \texttt{v}\ f(X) & :\!- & exists(X). \\
t(X)\ \texttt{v}\ f(X) & :\!- & forall(X). \\
t(X) & :\!- & w, forall(X). \\
f(X) & :\!- & w, forall(X). \\
w & :\!- & conjunct(X,Y,Z,Na,Nb,Nc), \\
 & & t(X), t(Y), t(Z), f(Na), f(Nb), f(Nc). \\
t(true). & & f(false). \qquad :\!-\ \texttt{not}\ w.
\end{array}$$

A QBF instance $\Phi = \exists X \forall Y \phi$ is encoded by the following set F_Φ of facts:



Legend: ASSAT: $\times\cdots\times$ SMODELS: $\circ\cdots\circ$, DLV: $*$—$*$

Figure 3: Reachability (REACH) and Same Generation (SAMEGEN)

- $exists(v)$, for each existential variable $v \in X$;
- $forall(v)$, for each universal variable $v \in Y$; and
- $conjunct(x_1, x_2, x_3, y_1, y_2, y_3)$, for each conjunct $l_1 \wedge l_2 \wedge l_3$ in ϕ, where (i) if l_i is a positive atom l_i then $x_i = v_i$, otherwise $x_i =$ "true", and (ii) if l_i is a negated atom $\neg v_i$, then $y_i = v_i$, otherwise $x_i =$ "false",

and the QBF instance Φ is valid if and only if $\mathcal{P}_{QBF} \cup F_\Phi$ has a stable model.

The encoding for SMODELS is essentially the same we used for **DLV**; we just had to rewrite the first two rules as follows:

$$\{true(X), false(X)\} \text{ :- } exists(X).$$
$$\{true(X), false(X)\} \text{ :- } forall(X).$$

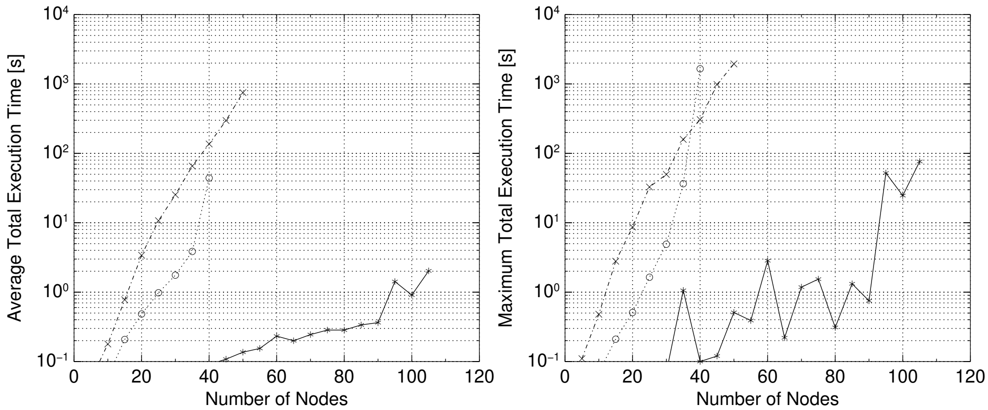
We randomly generated data sets where the number of $\forall$-variables is equal to the number of $\exists$-variables (that is, $|X| = |Y|$) and each conjunct contains at least two universal variables.

For HAMPATH, TSP, STRATCOMP, and QBF we generated 50 random instances for each problem size as indicated in the respective descriptions. For the more "deterministic" benchmarks REACH and SAMEGEN we used one instance per problem size.

In any case we allowed a maximum running time of 7200 seconds (two hours) and a maximum memory usage of 256MB. In the graphs displaying the benchmark results, the line of a system stops whenever some problem instance was not solved within this time or memory constraints.

6.2 Results and Discussion

In Figure 3 we show the results obtained for REACH and SAMEGEN problems; in Figure 4 we show the results obtained for HAMPATH; in Figure 5 we show the results for Ramsey,



Average time Maximum time

Legend: ASSAT: $\times \cdots \times$ SMODELS: $\circ \cdots \circ$, DLV: $*$—$*$

Figure 4: Hamiltonian Path (HAMPATH)

in Figure 6 those for STRATCOMP, and, finally, in Figure 7 the results for QBF are illustrated.

On average, on the set of proposed problems, SMODELS and ASSAT turn out to be significantly slower that **DLV**.

On the deductive database applications (REACH and SAMEGEN problems), this is caused by the fact that Lparse - the instantiator used by SMODELS and ASSAT - does not employ any database optimization technique and its instantiation procedure requires each variable in the program to be bound over a pre-defined domain. These results clearly show that **DLV** is better suited than SMODELS and ASSAT on database-oriented applications; the database optimization techniques employed by **DLV** pay off in this class of applications.

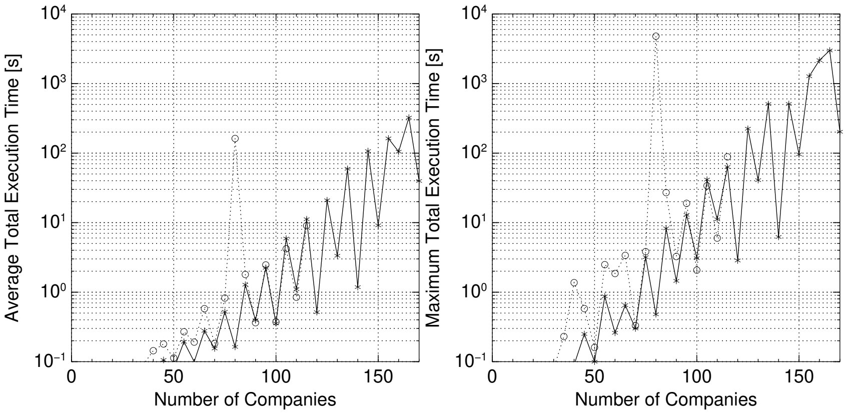
For Ramsey, the situation is somewhat more differentiated. Lparse is able to process Ramsey4-4, sizes 13 and 14, in less than one second, and the majority of time is consumed by SMODELS and ASSAT themselves. For size 15 (Ramsey3-7), Lparse still is very fast, and SMODELS only consumes about 18MB of memory, but does not terminate within 12 hours, whereas **DLV**'s Model Generator is able to solve this instance in less than once second. For all instances of Ramsey3-6, Ramsey3-7, and Ramsey4-5, Lparse requires (much) more than 256MB of memory and terminates after about one minute when this limit is reached. In comparison, **DLV** is able to instantiate Ramsey3-7, size 23, in about 200 seconds consuming 79MB of memory.

On the smallest instances of the STRATCOMP problem, SMODELS and **DLV** show similar performance, but **DLV** scales much better than SMODELS[8]. Indeed, the latter system stops much earlier than **DLV**. This is no surprise, as the treatment of disjunction in **DLV** is built-in; while in SMODELS disjunction is treated by a rewriting technique.

[8]Note that SMODELS does not provide a native support for disjunction. To solve the Σ_2^P-complete problems QBF and STRATCOMP, where disjunction is strictly needed, we have used the disjunctive extension GnT of SMODELS

Problem	Size	**DLV**	SMODELS	ASSAT
RAMSEY3-3	5	0.01	0.01	0.05
RAMSEY3-3	6	0.01	0.02	0.04
RAMSEY3-4	8	0.02	0.23	0.19
RAMSEY3-4	9	0.88	0.94	1.56
RAMSEY3-5	12	0.19	20.53	19.06
RAMSEY3-5	13	0.32	30.76	35.74
RAMSEY3-5	14	-	-	-
RAMSEY3-6	16	3.50	-	-
RAMSEY3-6	17	6.35	-	-
RAMSEY3-6	18	-	-	-
RAMSEY3-7	15	3.98	-	-
RAMSEY3-7	16	7.40	-	-
RAMSEY3-7	17	13.47	-	-
RAMSEY3-7	18	24.17	-	-
RAMSEY3-7	19	42.56	-	-
RAMSEY3-7	20	68.64	-	-
RAMSEY3-7	21	111.95	-	-
RAMSEY3-7	22	1513.00	-	-
RAMSEY3-7	23	-	-	-
RAMSEY4-4	13	0.21	70.12	2.58
RAMSEY4-4	14	0.31	109.69	3.83
RAMSEY4-4	15	0.45	-	5.61
RAMSEY4-4	16	0.80	-	12.88
RAMSEY4-4	17	4510.36	-	20.67
RAMSEY4-4	18	-	-	-
RAMSEY4-5	17	2.41	-	-
RAMSEY4-5	18	3.59	-	-
RAMSEY4-5	19	5.05	-	-
RAMSEY4-5	20	7.13	-	-
RAMSEY4-5	21	10.03	-	-
RAMSEY4-5	22	14.05	-	-
RAMSEY4-5	23	53.71	-	-
RAMSEY4-5	24	-	-	-

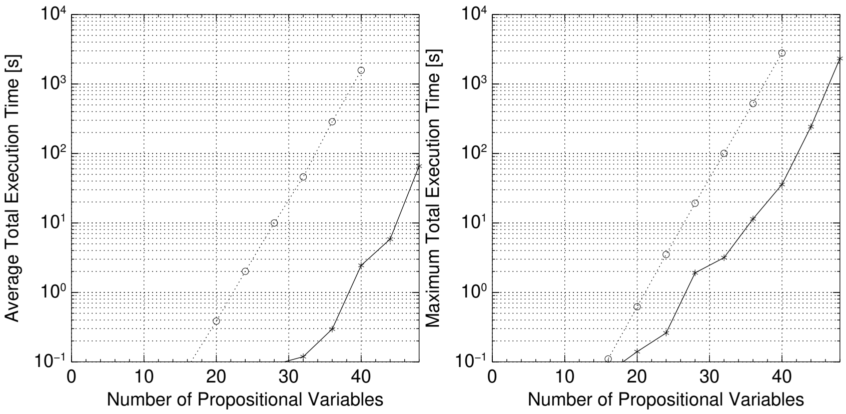
Figure 5: Ramsey (times in seconds)



Average time Maximum time

Legend: SMODELS: ∘ ··· ∘, DLV: ∗—∗

Figure 6: Strategic Companies (STRATCOMP)



Average time Maximum time

Legend: SMODELS: ∘ ··· ∘, DLV: ∗—∗

Figure 7: QBF

Thus, one expects that **DLV** is more efficient than SMODELS on Σ_2^P-hard problems (which require to deal efficiently with disjunction). A similar reasoning applies to QBF, where **DLV** performs clearly better.

The results on HAMPATH are instead more surprising. The systems SMODELS and ASSAT are somehow "specialized" for NP-complete problems, and we are aware that SMODELS is rather efficient on several NP-complete problems. The comparatively better performance of **DLV** over SMODELS on HAMPATH is probably due to a bad behaviour of SMODELS' heuristic on the benchmarked problem instances. We observe, however, that SMODELS behaves on average better than ASSAT on HAMPATH. This is probably due to the fact that the adopted encoding can be better exploited by the ASP systems employing the well-founded pruning operator, like **DLV** and SMODELS.

7 Recommendations and conclusions

Although deductive database systems are able to deal with larger amounts of data compared to ASP systems, they do not provide proper tools for handling the problems arising in data integration tasks. Indeed, the limited capability to manage integrity constraints, the inability to express and, therefore, answer computationally hard queries, and the inappropriateness in handling data incompleteness and inconsistencies, make deductive database systems unsuited for solving complex data integration problems.

The high expressiveness of ASP systems allows them to deal with the problems outlined above, as pointed out also in [51, 50].

Thus, even if ASP systems are able to deal with smaller input data than deductive database systems, they seem to be better suited to perform data integration tasks. Among the systems in this class, it is widely accepted that the two most relevant ASP systems which are nowadays available are **DLV** and SMODELS. SMODELS is more suitable for applications specific to Artificial Intelligence rather than for database-oriented applications. This is because SMODELS does not employ any database optimization technique and the instantiation procedure of SMODELS requires each variable in the program to be bound over a pre-defined domain. This requirement appears too restrictive in database applications; indeed, in database contexts, variables are supposed to vary in a wide range of values and, in some cases, it is not possible to fix the range of values a variable might assume. This intuition is confirmed by the fact that SMODELS turns out to be significantly slower than DLV on typical deductive database applications.

As far as ASSAT - another recent ASP system - is concerned, since it uses the same instantiation module of SMODELS, it has the same SMODELS limitations from a database perspective and does not seem to be well suited for data integration tasks.

In sum, from the analysis we carried out, it turns out that no computational logic system supports completely all features which are desirable for exploitation in the process of data integration. However, the ASP systems seem to be more appropriate than deductive database systems for data integration. Among the ASP systems, **DLV** seems to be the best-suited computational logic system to be incorporated in a data integration platform, since it is able to solve the computationally hard problems arising in the context of data integration, while showing better performance than the other ASP systems over large input data. However, some enhancements to this system are needed to make it more suited for data integration tasks. Among them, the implementation of some program rewriting techniques ("à la Magic Set") would be important in order to improve the efficiency of the system on larger input data. The implementation of suitable interfaces to relational databases (ODBC or similar) is also important. From the linguistic viewpoint the lack of constructs for expressing aggregate queries should also be overcome.

References

[1] A. Aggoun, D. Chan, P. Dufresne, et al. Eclipse user manual release 5.0, 2000.

[2] C. Anger, K. Konczak, and T. Linke. `NoMoRe`: A system for non-monotonic reasoning under answer set semantics. In W. Faber T. Eiter and M. Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, pages 406–410. Springer, 2001.

[3] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, California, 1988.

[4] C. Aravidan, J. Dix, and I. Niemela. Dislop: A research project on disjunctive logic programming. *AICommunications*, 10(3/4):151–165, 1997.

[5] C. Aravidan, J. Dix, and I. Niemela. Dislop: Towards a disjunctive logic programming system. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, pages 342–353. Springer, LNAI, 1997.

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

[7] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of the Conference on Management of Data (SIGMOD'86)*, pages 16–52, Washington, D.C., 1986. ACM Press.

[8] C. Batini and M. Lenzerini. A methodology for data schema integration in the entity relationship model. *IEEE Transactions on Software Engineering*, 10(6):650–664, 1984.

[9] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. *J. Logic Programming*, 10(3/4), 1991.

[10] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Logic Programming*, 10(3/4):255–299, 1991.

[11] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[12] Robert Bihlmeyer, Wolfgang Faber, Christoph Koch, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. `dlv` – an overview. In Uwe Egly and Hans Tompits, editors, *Proceedings of the 13th Workshop on Logic Programming (WLP'98)*, pages 65–67, Vienna, Austria, October 1998.

[13] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[14] Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, May/June 1997.

[15] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data integration under integrity constraints. In *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002*, pages 262–279, Toronto, Canada, 2002. Lecture Notes in Computer Science.

[16] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases.* Springer Verlag, New York, NY, 1990.

[17] Weidong Chen and David Scott Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

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

[19] P. Cholewinski, A. Marek, V.W. Mikitiuk, and M. Truszczynski. Computing with default logic. *Journal of Artificial Intelligence*, 112(1/2):105–146, 1999.

[20] Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The `dlv` System: Model Generator and Application Frontends. In F. Bry, B. Freitag, and D. Seipel, editors, *Proceedings of the 12th Workshop on Logic Programming (WLP'97), Research Report PMS-FB10*, pages 128–137, München, Germany, September 1997. LMU München.

[21] H. Decker. Integrity enforcement on deductive databases. In *Proc. of 1st Int. Conf. on Expert Database Systems*, 1986.

[22] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, and Gerald Pfeifer. System Description: `DLV`. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic*

Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings, number 2173 in Lecture Notes in AI (LNAI), pages 409–412. Springer Verlag, September 2001.

[23] M. Derr, S. Morishita, and G. Phipps. Design and implementation of the glue-nail database system. In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 147–167, Washington, DC, 1993.

[24] D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, pages 138–153. Springer Verlag, LNAI 2174, 2001.

[25] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[26] T. Eiter, G. Gottlob, and H. Mannila. Adding Disjunction to Datalog. In *Proceedings of the Thirteenth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-94)*, pages 267–278. ACM Press, May 1994.

[27] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

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

[29] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[30] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pushing Goal Derivation in DLP Computations. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 177–191, El Paso, Texas, USA, December 1999. Springer Verlag.

[31] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[32] B. Freitag, H. Shutz, and G. Specht. LOLA - a logic language for deductive databases and its implementation. In *Proc. of the 2nd International Symposium on Database Systems for Advanced Applications (DASFAA'91)*, pages 216–225, Tokyo, Japan, 1991.

[33] H. Gallaire. Impacts of logic on databases. In *Proc. of the International Conference on Very Large Databases*, pages 248–259, Cannes, France, 1981. IEEE Computer Society.

[34] H. Gallaire and J. Minker. *Logic and Databases.* Plenum Press, New York, 1978.

[35] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and databases: a deductive approach. *ACM Comput. Surv.*, 16(2):153–186, 1984.

[36] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Foundation of Deductive Databases and Logic Programming*, pages 149–176, 1988.

[37] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of The association for Computing Machinery*, 38(3):620–650, 1991.

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

[39] Ian Gent and Toby Walsh. The QSAT Phase Transition. In *Proceedings of the 16th AAAI*, 1999.

[40] J. Grant and J. Minker. The impact of logic programming on databases. *Commun. ACM*, 35(3):66–81, 1992.

[41] J. Han, L. Liu, and Z. Xie. Logicbase: a deductive database system prototype. In *Proc. of the 3rd International Conference on Knowledge Management (CIKM '94)*, pages 226–233, Gaithersburg, Maryland, 1994. ACM Press.

[42] Y. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *Proc. of 14th International Conference on Very Large Data Bases*, pages 382–394, Los Angeles, CA, 1988. Morgan Kaufmann.

[43] B. Jacobs. On database logic. *J. ACM*, 2 (Apr.):310–332, 1982.

[44] Tomi Janhunen, Ilkka Niemela, Patrik Simons, and Jia-Huai You. Partiality and disjunctions in stable model semantics. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 411–419. Morgan Kaufmann Publishers, Inc., 2000.

[45] B. Jiang. A suitable algorithm for computing partial transitive closures. In *Proc. IEEE Sixth International Conference on Data Engineering*, pages 264–271, Kobe, Japan, 1990. IEEE Computer Society.

[46] Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 359–363, 1992.

[47] W. Kiebling, H. Shmidth, W. Straub, and G. Dunzinger. DECLARE and SDS: Early efforts to commercialize deductive database technology. *VLDB J.*, 3(2):211–243, 1994.

[48] Donald E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing.* ACM Press, New York, 1994.

[49] A. Lefebvre and L. Vieille. On deductive query evaluation in the dedgin* system. In *1st International Conference on Deductive and Object Oriented Databases*, 1989.

[50] D. Lembo, M. Lenzerini, and R. Rosati. Integrating inconsistent and incomplete data sources. In *SEBD-02*, pages 299–306, 2002.

[51] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *KRDB-02*. CEUR Electronic Workshop Proceedings url: http://ceur-ws.org/Vol-54, 2002.

[52] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112, June 1997.

[53] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of International Conference on Very Large Data Bases (VLDB'96)*, pages 251–262, Bombay, India, 1996. Morgan Kaufmann.

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

[55] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[56] F. Lin and Y. Zhao. Assat: Computing answer sets of a logic program by sat solvers. *AAAI-02*, To appear, 2002.

[57] T. Linke. Graph theoretical characterization and computation of answer sets. pages 641–645, 2001.

[58] T. Linke, C. Anger, and K. Konczak. More on nomore. In G. Ianni and S. Flesca, editors, *Eighth European Workshop on Logics in Artificial Intelligence (JELIA'02)*, volume 2424, 2002.

[59] J.W. Lloyd. *Foundation of Logic Programming.* Springer Verlag, New York, NY, 1987.

[60] J. Lobo, J. Minker, and A. Rajasekar. *Foundation of Disjunctive Logic Programming.* Mit Press, 1992.

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

[62] D.W. Loveland. Near-horn PROLOG. In *4th International Cpnference on Logic Programming (ICLP'87)*, pages 456–459, 1987.

[63] D. Maier. *The theory of Relational Databases.* Computer Science Press, New York, 1983.

[64] H. Mannila and K.J. Raiha. *The design of relational databases.* Addison Wesley Publishing Company, Reading, Massachusetts, USA, 1991.

[65] S. Morishita, M. Derr, and G. Phipps. Design and implementation of the glue-nail database system. In *Proc. of ACM-SIGMOD'93 Conference*, pages 147–167, 1993.

[66] K. Morris, J. Ullman, and A. Van Gelder. Design overview of the nail! system. In *Proc. of Third International Conference on Logic Programming*, pages 554–568, London, UK, 1986. Springer Verlag.

[67] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. *ACM Trans. Database Systems*, 21(1):107–155, 1996.

[68] M.Werner. davinci v2.1.x online documentation, 1998.

[69] Ilkka Niemelä and Patrik Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.

[70] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A System for Answer Set Programming. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.

[71] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[72] G. Phipps, M. Derr, and K. Ross. Glue-nail: A deductive database system. In *Proc. of ACM-SIGMOD'91 Conference*, 1991.

[73] S. Potamianos and M. Stonebraker. The postgres rules system. *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 43–61, 1996.

[74] R. Ramakrishnan. *Application of Logic Databases.* Kluwer Accademic Publisher, Hingam, MA, 1994.

[75] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *VLDB J.*, 3(2):161–210, 1994.

[76] R. Ramakrishnan and J.D. Ullman. A survey of deductive database systems. *J. of Logic Programming*, 23(2):125–150, 1995.

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

[78] K. Ross. Modular stratification and magic sets for datalog programs with negation. In *Proc. of the ACM Symposium on Pronciples of Database Systems*, 1990.

[79] D. Saccà and C. Zaniolo. On the implementation of a simple class of logic queries. In *Proc. of the ACM Symposium on Pronciples of Database Systems*, 1986.

[80] D. Saccà and C. Zaniolo. Magic counting methods. In *Proc. of the ACM SIGMOD Annual Conference on Management of Data (SIGMOD '87)*, pages 49–59, San Francisco, CA, 1987. ACM Press.

[81] D. Saccà and C. Zaniolo. The generalized counting method for recursive logic queries. *Theoretical Computer Science*, 62, 1988.

[82] K. Sagonas, T. Swift, and D. Warren. XSB as an efficient deductive database engine. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*, pages 442–453, Minneapolis, Minnesota, 1994.

[83] Dietmar Seipel and Helmut Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In Antoni Olivé, editor, *Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94)*, pages 325–343. Universitat Politecnica de Catalunya (UPC), 1994.

[84] Patrik Simons. Extending the Stable Model Semantics with More Expressive Rules. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 305–316, El Paso, Texas, USA, December 1999. Springer Verlag.

[85] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[86] S. Tsur and C. Zaniolo. LDL: a logic bases data-language. In *Proc. of the 12th International Conference on Very Large Databases*, pages 33–41, Kyoto, Japan, 1986. VLDB Endowment, Berkley, CA.

[87] J. Ullman. *Principles of Database Systems*. Computer Science Press, New York, 1982.

[88] J. Ullman. Bottom-up beats top-down for datalog. In *Proc. of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems (PODS '89)*, pages 140–149, Philadelphia, PA, 1989. ACM Press.

[89] J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, New York, 1989.

[90] J.D. Ullman. Information integration using logical views. In *Proc. of International Conference on Database Theory (ICDT'97)*, pages 19–40, Delphi, Greece, 1997. Lecture Notes in Computer Science, Springer-Verlag.

[91] J. Vaghani, K. Ramanohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask, and J. Harland. The ADITI deductive database system. *VLDB J.*, 3(2):245–288, 1994.

[92] L. Vieille. Recursive axioms in deductive databases: the query/subquery approach. In *Proc. of 1st International Conference on Expert Database Systems*, 1986.

[93] L. Vieille. Database-complete proof procedures based on sld resolutions. In *Proc. of 4th International Conference on Logic Programming*, 1987.

[94] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69, 1989.

[95] L. Vieille, P. Bayer, and V. Kuechenhoff. *Integrity checking and materialized view handling by update propagation in the EKS-V1 system.* In "Materialized Views". Mit Press, Cambridge, MA, 1996.

[96] L. Vieille, P. Bayer, V. Kuechenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI'90 Workshop on Knowledge base Management Systems*, 1990.

[97] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[98] J. Wielemaker. Swi-prolog 3.4.3 reference manual, 1990–2000.