Università degli Studi della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XX Ciclo

Settore Disciplinare INF/01 INFORMATICA

Tesi di Dottorato

DLV^{DB} An ASP System for Data Intensive Applications

Claudio Panetta

Supervisori	**Coordinatore**
Prof. Nicola Leone	Prof. Nicola Leone
Dott. Giorgio Terracina	

Anno Accademico 2007 - 2008

DLV^{DB}

An ASP System for Data Intensive Applications

Ph.D. Thesis

Claudio Panetta

Sommario

La rapida crescita di sistemi informatici derivanti dalle diverse applicazioni cui Internet si presta, ha rapidamente aumentato la quantità di dati e di informazioni disponibili per l'elaborazione. In particolare, l'affermarsi del commercio elettronico, il diffondersi di sistemi per l'e-government delle pubbliche amministrazioni, l'ormai avviato processo di digitalizzazioni degli archivi e dei documenti in essi contenuti, la disponibilità di database medici sempre più completi e ricchi di informazioni e, più in generale, il sempre maggiore utilizzo dei sistemi informatici per la gestione strutturata di grandi quantità di dati hanno evidenziato l'urgenza di sviluppare nuove tecnologie che consentano di elaborare automaticamente ed efficientemente la quantità di dati derivante da questi settori emergenti.

Uno degli utilizzi principali dei sistemi di basi di dati (DBMS) consiste nella memorizzazione e nel recupero efficiente di grandi quantità di dati. L'elaborazione di tali informazioni, specialmente quella finalizzata all'estrazione di nuova conoscenza, è ormai riconosciuta come una tematica di ricerca di fondamentale importanza sia nell'ambito delle basi di dati, sia nell'ambito della ricerca industriale, in quanto offre grandi opportunità di sviluppo. In tale scenario, applicazioni come "Data Mining", "Data Werehousing" e "Online Analytical Processing (OLAP)" hanno ulteriormente evidenziato la necessità di sviluppare sistemi di basi di dati che supportino linguaggi maggiormente espressivi, in grado di consentire elaborazioni sempre più raffinate delle informazioni contenute nei Database. Il complesso di tali esigenze ha portato alla definizione di diverse estensioni per i modelli di rappresentazione dei dati (Modelli Relazionali basati sul concetto degli Oggetti), nonché alla definizione di nuovi costrutti sintattici (ricorsione e costrutti OLAP), ed all'estenzione dei DBMS (DataBase Management Systems) con linguaggi di programmazione di alto livello, basati su UDF (User Defined Functions).

Purtroppo, però anche i migliori sistemi di basi di dati attualmente in commercio non sono sufficientemente potenti e generali da poter essere efficacemente utilizzati per risolvere molte delle emergenti applicazioni. In generale, gli attuali DBMS non contengono i meccanismi di ragionamento necessari per estrarre conoscenza complessa dai dati disponibili. Tali meccanismi, dovrebbero essere in grado sia di gestire grandi quantità di informazioni, sia di realizzare sofisticati processi di inferenza sui dati per trarne nuove conclusioni.

Le capacità di ragionamento necessarie a tale scopo possono essere fornite dai sistemi basati su linguaggi logici. La Programmazione Logica Disgiuntiva (DLP) è un formalismo che consente di rappresentare, in maniera semplice e naturale, forme di ragionamento non monotono, planning, problemi diagnostici e, più in generale, problemi di elevata complessità computazionale. In DLP, un programma è una collezione di regole logiche in cui è consentito l'uso della disgiunzione nella testa delle regole e la negazione nel corpo. Una delle possibili semantiche per tali programmi è basata sulla nozione di modello stabile (*answer set*). Ad ogni programma viene associato un insieme di answer set, ognuno corrispondente ad una possibile visione del dominio modellato.

La DLP sotto tale semantica viene comunemente riferita con il termine di *Answer Set Programming* (ASP). Il recente sviluppo di efficienti sistemi basati sulla programmazione logica come DLV [80], Smodels [101], XSB [114], ASSAT [84, 86], Cmodels [62, 61], CLASP [56], etc., ha rinnovato l'interesse nei campi del ragionamento non-monotono e della programmazione logica dichiarativa per la risoluzione di molti problemi in differenti aree applicative. Conseguentemente, tali sistemi possono fornire le funzionalità di inferenza e ragionamento richieste dalle nuove aree di applicazione che interessano i sistemi di basi di dati.

Tuttavia, i sistemi basati sulla programmazione logica presentano notevoli limitazioni nella gestione di grandi quantità di dati non essendo dotati dell'opportuna tecnologia per rendere efficiente la loro gestione poiché eseguono le loro elaborazioni facendo uso di strutture dati gestite direttamente in memoria centrale. Inoltre, la maggior parte delle applicazioni di interesse comune coinvolge grandi moli di dati su cui applicare complessi algoritmi di inferenza logica difficilmente elaborabili sia dai sistemi di programmazione logica, sia dai tradizionali database.

Queste considerazioni mettono in evidenza la necessità di tecniche efficienti ed efficaci che combinino le qualità dei sistemi di inferenza logica con quelle dei sistemi di gestione delle basi di dati. In letteratura, le proposte di soluzione a tale problema sono culminate nei Sistemi di Basi di Dati Deduttive (DDS) [25, 52, 23, 63], che combinano le due realtà dei sistemi logici e dei DBMS. In pratica, i DDS sono il risultato di una serie di tentativi di adattare i sistemi logici, che hanno una visione del mondo basata su pochi dati, ad applicazioni su grandi moli di dati attraverso interazioni intelligenti con le basi di dati. In particolare, i DDS sono forme avanzate di DBMS i cui linguaggi di interrogazione, basati sulla logica, sono molto espressivi. I DDS non memorizzano solo le informazioni esplicite in un database relazionale, ma memorizzano anche regole che consentono inferenze deduttive sui dati memorizzati. L'uso congiunto di tecniche sviluppate nell'ambito delle basi di dati relazionali con quelle della programmazione logica dichiarativa, consente in linea di principio ai DDS di realizzare ragionamenti complessi su grandi quantità di dati.

Tuttavia, nonostante le loro potenzialità lo sviluppo di sistemi DDS a livello industriale non ha ricevuto molta attenzione. Ciò principalmente è stato dovuto al fatto che è estremamente complesso ottenere sistemi particolarmente efficienti ed efficaci; infatti, le attuali implementazioni di DDS sono basate su due approcci estremi: uno basato sul miglioramento dell'elaborazione dei dati da parte dei sistemi logici, l'altro basato sull'aggiunta di capacità di ragionamento ai DBMS (ad esempio tramite l'uso di SQL99, o di funzioni esterne). Entrambi tali approcci presentano limitazioni importanti. In particolare, i DDS basati sulla logica possono gestire una quantità limitata di dati, dal momento che, gli attuali sistemi logici eseguono i loro ragionamenti direttamente in memoria centrale; inoltre, essi forniscono interoperabilità limitate con DBMS esterni. Al contrario, i DDS basati sui database offrono funzionalità avanzate di gestione dei dati, ma scarse capacità di ragionamento (sia a causa della poca espressività dei linguaggi di interrogazione, sia a causa di problemi di efficienza).

Riassumendo, possiamo affermare che:

- Gli attuali sistemi di basi di dati implementano moduli sufficientemente robusti e flessibili capaci di gestire grandi quantità di dati, ma non possiedono un linguaggio sufficientemente espressivo da consentire ragionamenti complessi su questi dati.
- I sistemi basati sulla programmazione logica, possiedono elevate capacità di ragionamento e sono in grado di modellare e risolvere con facilità problemi di elevata complessità ma presentano notevoli limitazioni nella gestione di grandi quantità di dati poiché eseguono le loro elaborazioni facendo uso di strutture dati gestite direttamente in memoria centrale.

- I sistemi di basi di dati deduttive consentono di gestire i dati memorizzati su DBMS, ma, dal momento che, eseguono i loro ragionamenti direttamente in memoria centrale, possono gestire una quantità limitata di dati;

Dalle precedenti osservazioni, si evidenzia la necessità di realizzare applicazioni che combinino il potere espressivo dei sistemi di programmazione logica con l'efficiente gestione dei dati tipica dei database.

Il contributo di questa tesi si colloca nell'area della ricerca sulle basi di dati deduttive con l'obiettivo di colmare il divario esistente tra sistemi logici e DBMS. In questa tesi viene descritto un nuovo sistema, DLV^{DB}, che ha la caratteristica di possedere le capacità di elaborazione dati desiderabili da un DDS ma di supportare anche le funzionalità di ragionamento più avanzate dei sistemi basati sulla programmazione logica disgiuntiva.

DLV^{DB} è stato progettato come estensione del sistema DLV e combina l'esperienza maturata nell'ambito del progetto DLV nell'ottimizzare programmi logici con le avanzate capacità di gestione dei dati implementate nei DBMS esistenti. Ciò consente di applicare tale sistema in ambiti che necessitano sia di valutare programmi complessi, sia di lavorare su grandi quantità di dati. DLV^{DB} è in grado di fornire, così sostanziali miglioramenti sia nelle prestazioni relative alla valutazione dei programmi logici, sia nella facilità di gestione dei dati di input e di output possibilmente distribuiti su più database. L'interazione con le basi di dati è realizzata per mezzo di connessioni ODBC che consentono di gestire in modo piuttosto semplice dati distribuiti su vari database in rete. DLV^{DB} consente di applicare diverse tecniche di ottimizzazione sviluppate sia nell'ambito dei sistemi logici, come ad esempio i magic set, sia nell'ambito della gestione delle basi di dati, come ad esempio tecniche di join ordering, inoltre sono stati integrati nel sistema i predicati per l'aggregazione di DLV (count, min, max, avg, sum) che avvicinano il linguaggio alle potenzialità di SQL, ma anche la possibilità di integrare nel programma logico, per natura dichiarativo, chiamate a funzioni esterne sviluppate con tecniche procedurali; ciò rende possibile integrare aspetti dichiarativi ed aspetti procedurali di un problema in un'unica framework. Infine, per consentire la gestione di tipi di dati con strutture ricorsive (es. XML) si è introdotta la possibilità di gestire liste di elementi, eventualmente innestate, nel programma logico.

Inoltre in questa tesi viene presentata l'attività di analisi di tipo sperimentale effettuata al fine di valutare le prestazioni di DLV^{DB}, soprattutto in riferimento a velocità di esecuzione di query e quantità di dati gestibili. Questi test hanno dimostrato come il sistema apporta numerosi vantaggi rispetto ai sistemi esistenti, sia in termini di tempi di esecuzione delle query, sia in termini di quantità di dati che esso riesce a gestire contemporaneamente.

In sintesi, i contributi di questo lavoro possono essere riassunti come segue:

- Sviluppo di un sistema in grado di fondere il potere espressivo dei sistemi ASP con l'efficiente gestione dei dati offerta dagli attuali Database;
- Sviluppo di una strategia di valutazione dei programmi logici in grado di minimizzare l'utilizzo della memoria centrale massimizzando l'utilizzo delle tecnologie implementate dai DBMS;
- Estensione del linguaggio DLP mediante l'introduzione di chiamate a funzioni esterne e il supporto a tipi di dati con strutture ricorsive come le liste;
- Realizzazione di un'analisi comparativa tra le prestazioni offerte da DLV^{DB} e le prestazioni dei sistemi esistenti.

Contents

Chapter 1

Introduction

Current capabilities of generating and collecting data are increasing rapidly. The wide-spread use of internet applications for most commercial activities, the computerization of many business and government transactions, and the advances in data collection tools have provided us with huge amounts of data. This explosive growth in data and databases has generated an urgent need for new techniques and tools that can intelligently and automatically infer useful information and knowledge from available data.

One of the most fundamental uses of a Database is to store and retrieve information, particularly when there is a large amount of data to be stored. Mining information and knowledge from large databases has been recognized by many researchers as a key research topic in database systems and machine learning fields, and by many industrial companies as an important area with an opportunity of major revenues.

In this scenario, a mounting wave of data intensive and knowledge based applications, such as Data Mining, Data Warehousing and Online Analytical Processing (OLAP) have created a strong demand for more powerful database languages and systems. This led to the definition of both several data model extensions (e.g., the Object Relational model), and new language constructs (e.g., recursion and OLAP constructs), and various database extenders (based, e.g., on user defined functions), to enhance the current Database Management Systems (DBMSs). A great effort in this direction has been carried out with the introduction of a new standard for SQL, namely SQL99 [126] which provides, among other features, support to object oriented databases and recursive queries. However, the adoption of SQL99 is still far from being a “standard”; in fact almost all current DBMSs do not fully support it and, in some cases, they adopt proprietary (non standard) language constructs and functions to implement parts of it. Moreover, the efficiency of current implementations of SQL99 constructs and their expressiveness are still not sufficient for performing complex reasoning tasks on huge amounts of data. On the other hand, the explosive growth of new database applications has, in several cases, outpaced the progress made by database technology.

Knowledge representation and reasoning capabilities required in these contexts could be provided also by a powerful rule-based formalism like disjunctive logic programming (DLP). Disjunctive logic programming under answer set semantics (DLP, ASP) is a powerful rule-based formalism for knowledge representation and reasoning. The language of DLP is very expressive, and allows to model also advanced knowledge-based tasks arising in modern application-areas like, e.g., information integration and knowledge management. The recent development of efficient logic-based systems like DLV [80], Smodels [101], XSB [114], ASSAT [84, 86], Cmodels

[62, 61], CLASP [56], etc., has renewed the interest in the area of non-monotonic reasoning and declarative logic programming for solving real world problems in a number of application areas. As a consequence, they can provide the powerful reasoning capabilities needed to solve novel complex database problems. However, as previously pointed out, many of the interesting problems are "data intensive" and can not be handled in a typical logic programming system working in main-memory.

Examples of applications that can not be handled neither by traditional databases, due to the complex reasonings they require, nor by main-memory logic systems, due to the great amount of involved data, are: the automatic correction of census data [48]; the integration of inconsistent and incomplete data [79]; the elaboration on Scientific Databases in order to detect molecular features (e.g., gene functions [103]) or to analyze the medical histories in order to verify the possible causes of diseases (e.g., radioactivity or genome malformation).

The considerations above put into evidence that efficient and effective data management techniques combining Logic Inference Systems with Database Management Systems, are mandatory. In particular, there is the need of combining the expressive power of logic-based systems with the efficient data management features of DBMSs. Indeed, logic-based systems provide an expressive power that goes far beyond that of SQL99, whereas good DBMSs provide very efficient query optimization mechanisms allowing to handle massive amounts of data.

In the literature Deductive Database Systems (DDS) have been proposed to combine these two realities [25, 52, 23, 63]; basically, they are an attempt to adapt typical Datalog systems, which have a "smalldata" view of the world, to a "largedata" view of the world via intelligent interactions with some DBMSs. In more detail, DDSs are advanced forms of database management systems, whose query languages, based on logics, are very expressive. DDSs not only store explicit information in the manner of a relational database, but they also store rules that enable deductive inferences based on the stored data. Using techniques developed for relational systems in conjunction with declarative logic programming, deductive databases are capable of performing reasoning based on that information.

After an initial enthusiastic period of fervent research on DDSs, motivated by their great potential, the successive phase of development of practical systems did not receive the expected attention, mainly because of the high difficulties encountered to obtain efficient and effective implementations. This led to a period in which deductive databases have been considered "out of fashion" for many years.

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

However, recently emerging application contexts such as the ones raising from the natural recursion across nodes in the Internet, or from the success of intrinsically recursive languages such as XML [141], renewed the interest of the scientific community in the development of efficient and flexible deductive databases systems [1, 91].

Summarizing:

- Database systems are nowadays robust and flexible enough to efficiently handle large amounts of data, possibly distributed; however, their query languages are not sufficiently expressive to support reasoning tasks on such data.

- Logic-based systems are endowed with highly expressive languages, allowing them to support complex reasoning tasks, but they work in main-memory and, hence, can handle limited amounts of data.

- Deductive database systems allow to access and manage data stored in DBMSs, however they perform their computations mainly in main-memory and provide limited interoperability with external (and possibly distributed) DBMSs.

1.1 Objectives and contributions

This thesis provides a contribution in the setting outlined above, bridging the gap between logic-based systems and DBMSs. It presents a new system, named DLV^{DB}, having all the features of a DDS but supporting also all the features of state-of-the art logic systems. DLV^{DB} allows substantial improvements in both the evaluation of logic programs and the management, within an existing logic system, of input and output data distributed on several databases.

DLV^{DB} has been developed as an extension of the well known Answer Set Programming (ASP) system DLV and combines the experience in optimizing logic programs gained within the DLV project with the well assessed data management capabilities of existing DBMSs. This makes it well suited to be applied on both complex and data intensive applications.

The DLV^{DB} system provides a well established infrastructure for the interoperation with databases and allows the application of several optimization techniques already developed or under development in the DLV project (such as magic sets [12, 16, 98, 116, 118]). Interoperation with databases is provided by ODBC connections; these allow to handle, in a quite simple way, data residing on various databases over the network.

Moreover, the DLV^{DB} system presents an evaluation strategy devoted to carry out as much as possible of the reasoning tasks in mass memory without degrading performances, thus allowing to deal with data-intensive applications; it exemplifies the usage of DLP for those problems characterized by both declarative and procedural components, via the usage of external function calls yet improving efficiency; it extends the expressiveness of DLP for supporting also the management of recursive data structures (e.g., lists).

We carried out a thorough experimental analysis for comparing the performance of the DLV^{DB} system, with those of state-of-the-art systems (both logic-based and databases); the analysis is based on the application of the considered systems in three different contexts:

- Classical deductive problems (see [14, 64]) which, basically, require the execution of recursive queries on different kinds of data sets.

- Querying inconsistent and incomplete data; we exploited a data integration framework developed in the INFOMIX project (IST-2001-33570) [40] which integrates real data from a university context.

- Querying RDF(S) ontologies with answer set programming.

These experiments show that DLV^{DB} may provide both important speed ups (in some cases in order of magnitudes) in the running time and the capability to handle the largest amounts of data.

Summarizing, the overall contributions of this work are the following:

- The development of a fully fledged system enhancing in different ways the interactions between logic-based systems and DBMSs.
- The development of an efficient database-oriented, evaluation strategy for logic programs allowing to minimize the usage of main-memory and to maximize the advantages of optimization techniques implemented in existing DBMSs.
- Full support to disjunctive datalog with unstratified negation, and aggregate functions.
- Extension of DLP with external function calls, particularly suited for solving inherently procedural sub-tasks but also for improving knowledge-modelling power.
- Extension of DLP for supporting the management of recursive data structures.
- Primitives to integrate data from different databases.
- The execution of a thorough experimental comparative analysis of the performance of state-of-the-art systems and DLV^{DB}.

1.2 Plan of the work

The work are organized as follows: in Chapter 2 we recall some background notion which will be useful throughout the rest of the thesis. In Chapter 3 we present an overview of Declarative Computational Logic Systems, whereas in Chapter 4 we introduce the main features developed in DLV^{DB}. In Chapter 5 we describe system architectures and the evaluation strategie implemented in DLV^{DB}; in Chapter 6 we present three different data-intensive applications, whereas experimental results on these applications are presented in Chapter 7. Finally, in Chapter 8 we draw some conclusions.

Chapter 2

Disjunctive Datalog

Disjunctive Logic Programming (DLP) under the answer set semantics, evolved significantly during the last decade, and has been recognized as a convenient and powerful method for declarative knowledge representation and reasoning. Several systems supporting DLP have been implemented so far, thereby encouraging a number of applications in several real-world contexts ranging, e.g., from information integration, to frauds detection, to software configuration, and many others.

In this chapter we present the *Answer Set Programming* (ASP) framework based on a Disjunctive Logic Programming (DLP) language extended with aggregates and functions. In particular, we first define the syntax of this language and its associated semantics, i.e. the Answer Set Semantics. Then, we illustrate the usage of Answer Set Programming as a formalism for knowledge representation and reasoning.

2.1 Syntax

Following Prolog's convention, strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. In addition, DLV^{DB} also supports positive integer constants and arbitrary string constants, which are embedded in double quotes. A *term* is either a *simple term* or a *list term*. A *simple term* is either a avariable or a constant. A *list term* can be defined using the following two forms:

- $[t_1, \ldots, t_n]$ where $t_1, \ldots, t_n$ are terms;
- $[h|t]$ where h (the head of the list) is a term, and t (the tail of the list) is a list term.

An *atom* is either an *ordinary atom*, an *aggregate atom* or an *external atom*. An *ordinary 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; *external* and *aggregate atom* predicate names are conventionally preceded by "#". *External atoms* introduce function call in the program; we assume, by convention, that only the last variable O of an external atom $\#f(X_1, \ldots, X_n, O)$ is considered as an output parameter, while all the other variables must be intended as input for f.

Two special external atoms are reserved for lists manipulation, namely $\#$head(L,H), which receives a list L and returns its head H, and $\#$tail(L,T), which returns the tali T of L. Actually, our current implementation imposes some restrictions on the generic definition of list terms. In particular, in $[t_1, \ldots, t_n]$ only (possibly nested lists of) constants are allowed, whereas in $[h|t]$, h can be either a constant or a variable and t can be only a variable. Note that these restrictions, coupled with the availability of #head and #tail do not limit language expressiveness.

A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\mathit{Vars} : \mathit{Conj}\}$, where Vars is a list of variables and Conj is a conjunction of standard atoms.[1] A *ground set* is a set of pairs of the form $\langle \overline{t} : \mathit{Conj} \rangle$, where $\overline{t}$ is a list of constants and Conj is a ground (variable free) conjunction of standard atoms.

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

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

Example 2.1.1 *The following are aggregate atoms, where the latter contains a ground set and could be a ground instance of the former:*

$$\#\mathtt{max}\{Z : r(Z), a(Z,V)\} > Y$$
$$\#\mathtt{max}\{\langle 2 : r(2), a(2,k)\rangle, \langle 2 : r(2), a(2,c)\rangle\} > 1$$

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 $\mathrm{not}\ l$, where l is a classical literal; in the former case ℓ is *positive*, and in the latter case *negative*. Given a classical literal l, its *complementary* literal $\neg l$ 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.

Moreover, DLV^{DB} provides built-in predicates such as the comparative predicates equality, less-than, and greater-than ($=$, $<$, $>$) and arithmetic predicates like addition or multiplication. For details, we refer to [45].

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

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \ \texttt{:-}\ \ b_1, \cdots, b_k,\ \mathrm{not}\ b_{k+1}, \cdots,\ \mathrm{not}\ b_m. \qquad (2.1)$$

where $a_1, \ldots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are (ordinary, aggregate or external) atoms, $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \vee \ \cdots \ \vee \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k,\ \mathrm{not}\ b_{k+1}, ...,\ \mathrm{not}\ b_m$ 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. $k = m = 0$), it is called a fact, and we usually omit the " :- " sign.

If r is a rule of form (2.1), then $H(r) = \{a_1, \ldots, a_n\}$ is the set of the literals in the head and $B(r) = B^+(r) \cup B^-(r)$ 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* (alternatively, *disjunctive logic program*, *disjunctive deductive database*) $\mathcal{P}$ is a finite set of rules. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive*,[2] and a v-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *datalog program* (or *normal logic program*, *deductive database*).

[1] Intuitively, a symbolic set $\{X{:}a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, i.e., $\{X \,|\, \exists Y s.t.\ a(X,Y), p(Y)\ \mathit{is\ true}\}$.

[2] In positive programs negation as failure (not) does not occur, while strong negation ($\neg$) may be present.

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

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

Let the *level mapping* of a program $\mathcal{P}$ be a function $||\ ||$ from the predicates in $\mathcal{P}$ to finite ordinals.

A DLP *program* $\mathcal{P}$ is called *stratified*^{not} [5, 109], if there is a level mapping $||\ ||_s$ of $\mathcal{P}$ such that, for every rule r,

1. For any $l \in B^+(r)$, and for any $l' \in H(r)$, $||l||_s \leq ||l'||_s$;

2. For any $l \in B^-(r)$, and for any $l' \in H(r)$, $||l||_s < ||l'||_s$;

3. For any $l, l' \in H(r)$, $||l||_s = ||l'||_s$.

A DLP *program* $\mathcal{P}$ is called *stratified*^{aggr} [32], if there is a level mapping $||\ ||_a$ of $\mathcal{P}$ such that, for every rule r,

1. If l appears in the head of r, and l' appears in an aggregate atom in the body of r, then $||l'||_a < ||l||_a$; and

2. If l appears in the head of r, and l' occurs in a standard atom in the body of r, then $||l'||_a \leq ||l||_a$.

3. If both l and l' appear in the head of r, then $||l'||_a = ||l||_a$.

Example 2.1.2 Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:

$$\begin{array}{l} q(X) \text{ :- } p(X), \#\texttt{count}\{Y : a(Y,X), b(X)\} \leq 2. \\ p(X) \text{ :- } q(X), b(X). \end{array}$$

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

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

2.2 Answer Set Semantics

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

A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for* S. Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of* S is the following ground set of pairs $inst(S)$: $\{\langle\gamma(Vars) : \gamma(Conj)\rangle \mid \gamma$ *is a local substitution for* $S\}$.[3]
A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example 2.2.1 Consider the following program $\mathcal{P}_1$:

$$\begin{array}{l} q(1) \vee p(2,2). \\ q(2) \vee p(2,1). \\ t(X) \text{ :- } q(X), \#\texttt{sum}\{Y : p(X,Y)\} > 1. \end{array}$$

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

$$\begin{array}{l} q(1) \vee p(2,2). \\ t(1) \text{ :- } q(1), \#\texttt{sum}\{\langle 1 : p(1,1)\rangle, \langle 2 : p(1,2)\rangle\} > 1. \\ q(2) \vee p(2,1). \\ t(2) \text{ :- } q(2), \#\texttt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1. \end{array}$$

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

An interpretation also provides a meaning for aggregate literals.

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

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

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

We now recall the generalization of the Gelfond-Lifschitz transformation to programs with aggregates from [44].

[3]Given a substitution σ and a DLP object Obj (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

Definition 2.2.2 [44] *Given a ground* DLP *program $\mathcal{P}$ and a total interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting all rules in which a body literal is false w.r.t. I. I is an answer set of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.*

Example 2.2.3 Consider the following two programs:

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

$Ground(P_1) = \{p(a) \text{ :- } \#\texttt{count}\{\langle a : p(a)\rangle\} > 0.\}$ and $Ground(P_2) = \{p(a) \text{ :- } \#\texttt{count}\{\langle a : p(a)\rangle\} < 1.\}$; consider also interpretations $I_1 = \{p(a)\}$ and $I_2 = \emptyset$. Then, $Ground(P_1)^{I_1} = Ground(P_1)$, $Ground(P_1)^{I_2} = \emptyset$, and $Ground(P_2)^{I_1} = \emptyset$, $Ground(P_2)^{I_2} = Ground(P_2)$ hold.

I_2 is the only answer set of P_1 (because I_1 is not a minimal model of $Ground(P_1)^{I_1}$), whereas P_2 admits no answer set (I_1 is not a minimal model of $Ground(P_2)^{I_1}$, and I_2 is not a model of $Ground(P_2) = Ground(P_2)^{I_2}$).

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

List terms are handled by suitable function calls; in particular, programs containing list terms are automatically rewritten to contain only terms and function calls.

As an example, the rule *q(H):- dom(H), list(T), list([H|T]).* is translated into *q(H):-dom(H), list(T), list(L), #head(L,H), #tail(L,T).*

Recall that, given an external atom $\#f(X_1, \ldots, X_n, O)$ used in a rule r, only the last variable O can be considered as an output parameter, while all the other variables must be intended as input for f; this corresponds to the function call $f(X_1, \ldots, X_n) = O$.

As an example, consider the program:

$$person(id1, "John", "Smith").$$
$$mergedNames(ID, N) \text{ :- } person(ID, FN, LN), \#concat(FN, LN, N).$$

Ground program is:

$$person(id1, "John", "Smith").$$
$$mergedNames(id1, "JohnSmith") \text{ :- } person(id1, "John", "Smith").$$

Note that, since the grounding phase instantiates all the variables, there is no need to invoke again the functions associated with external atoms after the grounding. As a consequence, the handling of external atoms can be carried out completely during the grounding.

Function call introduce in the program new constant values (e.g., ”John Smith”), this can generate infinite domain. In order to avoid the generation of infinite-sized answer sets programs must be value-invention restricted (cfr. [22]), i.e. new values possibly introduced by external atoms must not propagate through recursion.

2.3 Knowledge Representation and Reasoning

Answer Set Programming has been proved to be a very effective formalism for *Knowledge Representation and Reasoning (KRR)*. It can be used to encode problems in a highly declarative fashion, following the "Guess&Check"(Guess/Check/Optimize) methodology presented in [36]. In this section, we first describe the Guess/Check/Optimize technique and we then illustrate how to apply it on a number of examples. Finally, we show how the modelling capability of ASP is significatively enhanced by supporting function symbols.

2.3.1 The Guess and Check Programming Methodology

Many problems, also problems of comparatively high computational complexity (Σ_2^P-complete and Δ_3^P-complete problems), can be solved in a natural manner by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so in a uniform way 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 two main parts:

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

Checking Part The (optional) checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program filters the solution candidates in such a way that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the admissible solutions for the problem instance I.

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., checking is completely integrated into the guessing part such that solution candidates are always solutions. Also, 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 further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

In general, both $\mathcal{G}$ and $\mathcal{C}$ may be arbitrary collections of rules, and it depends on the complexity of the problem at hand which kinds of rules are needed to realize these parts (in particular, the checking part).

For problems with complexity in NP, often a natural Guess/Check/Optimize program can be designed with the two parts clearly separated into the following simple layered structure:

- The guessing part $\mathcal{G}$ consists of disjunctive rules that "guess" a solution candidate S.
- The checking part $\mathcal{C}$ consists of integrity constraints that check the admissibility of S.

Each layer may have further auxiliary predicates, for local computations.

The disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches.

It is worth remarking that the Guess/Check/Optimize programming methodology has also positive implications from the Software Engineering viewpoint. Indeed, the modular program structure in Guess/Check/Optimize allows for developing programs incrementally, which is helpful to simplify testing and debugging. One can start by writing the guessing part $\mathcal{G}$ and testing that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one adds the checking part and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ encode the admissible solutions.

2.3.2 Applications of the Guess and Check Technique

In this section, we illustrate the declarative programming methodology described in Section2.3.1 by showing its application on a number of concrete examples.

Let us consider a classical NP-complete problem in graph theory, namely Hamiltonian Path.

Definition 2.3.1 [HAMPATH] *Given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once?*

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

$$\left.\begin{array}{ll}inPath(X,Y) \vee\ outPath(X,Y) \text{ :- } start(X), arc(X,Y). & \\ inPath(X,Y) \vee\ outPath(X,Y) \text{ :- } reached(X), arc(X,Y). & \\ reached(X) \text{ :- } inPath(Y,X). & \textbf{(aux.)}\end{array}\right\} \textbf{Guess}$$

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

The two disjunctive rules guess a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary predicate $reached$ is used, which is associated with the guessed predicate $inPath$ using the last rule. Note that $reached$ is completely determined by the guess for $inPath$, and no further guessing is needed.

In turn, through the second rule, the predicate $reached$ influences the guess of $inPath$, which is made somehow inductively. Initially, a guess on an arc leaving the starting node is made by the first rule, followed by repeated guesses of arcs leaving from reached nodes by the second rule, until all reached nodes have been handled.

In the checking part, the first two constraints ensure that the set of arcs S selected by $inPath$ meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S.

A less sophisticated encoding can be obtained by replacing the guessing part with the single rule

$$inPath(X,Y) \vee\ outPath(X,Y) \text{ :- } arc(X,Y).$$

that guesses for each arc whether it is in the path and by defining the predicate $reached$ in the checking part by rules

$$\begin{array}{l}reached(X) \text{ :- } start(X). \\ reached(X) \text{ :- } reached(Y), inPath(Y,X).\end{array}$$

However, this encoding is less preferable from a computational point of view, because it leads to a larger search space.

It is easy to see that any set of arcs S which satisfies all three constraints must contain the arcs of a path $v_0, v_1, \ldots, v_k$ in G that starts at node $v_0 = a$, 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 in fact a Hamiltonian Cycle (from which a Hamiltonian path can be immediately computed, by dropping the last arc).

Thus, given a set of facts $\mathcal{F}$ for $node$, arc, and $start$, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the corresponding graph has a Hamiltonian Path. The above program correctly encodes the decision problem of deciding whether a given graph admits a Hamiltonian Path or not.

This encoding is very flexible, and can be easily adapted to solve the *search problems* Hamiltonian Path and Hamiltonian Cycle (where the result has to be a tour, i.e., a closed path). If we want to be sure that the computed result is an *open* path (i.e., it is not a cycle), we can easily impose openness by adding a further constraint $:\!-\ start(Y), inPath(\_, Y).$ to the program (like in Prolog, the symbol '_' stands for an anonymous variable whose value is of no interest). Then, the set S of selected arcs in any answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at a. If, on the other hand, we want to compute the Hamiltonian cycles, then we just have to strip off the literal $\text{not}\ start(X)$ from the last constraint of the program.

In the previous examples, we have seen how a search problem can be encoded in a DLP program whose answer sets correspond to the problem solutions. We next see another use of the Guess/Check/Optimize programming technique. We build a DLP program whose answer sets witness that a property does not hold, i.e., the property at hand holds if and only if the DLP program has no answer set. Such a programming scheme is useful to prove the validity of co-NP or Π_2^P properties. We next apply the above programming scheme to a well-known problem of number and graph theory.

Definition 2.3.2 [RAMSEY] *The Ramsey number $R(k, m)$ is the least integer n such that, no matter how we color the arcs of the complete undirected graph (clique) with n nodes using two colors, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique).*

Ramsey numbers exist for all pairs of positive integers k and m [110]. We next show a program $\mathcal{P}_{ramsey}$ that allows us to decide whether a given integer n is <u>not</u> the Ramsey Number $R(3, 4)$. By varying the input number n, we can determine $R(3, 4)$, as described below. Let $\mathcal{F}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with n nodes. $\mathcal{P}_{ramsey}$ is the following Guess/Check/Optimize program:

$$\left.\begin{array}{l} blue(X,Y) \vee\ red(X,Y)\ :\!-\ arc(X,Y). \end{array}\right\}\ \textbf{Guess}$$

$$\left.\begin{array}{l} :\!-\ red(X,Y),\ red(X,Z),\ red(Y,Z). \\ :\!-\ blue(X,Y),\ blue(X,Z),\ blue(Y,Z), \\ \quad\ blue(X,W),\ blue(Y,W),\ blue(Z,W). \end{array}\right\}\ \textbf{Check}$$

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red clique (i.e., a complete graph) with 3 nodes, and the second constraint eliminates the colorings containing a blue clique with 4 nodes. The program $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has an answer set if and only if there is a coloring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular n, then n is <u>not</u> $R(3, 4)$, that is, $n < R(3, 4)$. On the other hand, if $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has no answer set, then $n \geq R(3, 4)$. Thus, the smallest n such that no answer set is found is the Ramsey number $R(3, 4)$.

2.3.3 Enhanced KRR Capabilities by Function Symbols

As shown by the previous examples, ASP is particularly well-suited for modelling and solving problems that involve common-sense reasoning as well as for advanced knowledge-based tasks. As a matter of fact, this paradigm has been successfully applied to a range of applications including information integration, software configuration, reasoning about actions and change, etc.

These applications have evidenced some limitations of ASP languages and systems, that should be overcome to make ASP better suited for real-world applications even in industry. While Answer Set Semantics, which underlies ASP, was defined in the setting of a general first order language, current ASP frameworks and implementations, like DLV [80], Smodels [124], clasp [56] and other efficient solvers, are based in essence on function-free languages and resort to Datalog with negation and its extensions. Therefore, even by using state-of-the-art systems, one cannot directly reason about recursive data structures and infinite domains, such as XML/HTML documents, lists, time, etc. This is a strong limitation, both for standard knowledge-based tasks and for emerging applications, such as those manipulating XML documents.

Since one is forced to work with finite domains, potentially infinite processes cannot be represented naturally in ASP. Additional tools to simulate unbounded domains must be used. A notable example is the DLVK [37] front-end of the DLV system which implements the action language K [38]. Constants are used to instantiate a sufficiently large domain (estimated by the user) for solving the problem; this may incur high space requirements, and does not scale to large instances. Another example is given by recursive data structures like lists that can be simulated only through unnatural encodings.

Function symbols, in turn, are a very convenient means for generating infinite domains and objects, and allow for a more natural representation of problems in such domains. Recursive data structures can be immediately represented, without resorting to indirect encodings. Besides, there is no need to use constants to bound variables whose maximum value is a priori unknown like, for instance, variables representing a time or a plan length.

Chapter 3

Comparisons to other Systems

This chapter is devoted to give a review of computational logic systems and to point out the applicability of them to data intensive application. In particular, in section 3.1 we focus on deductive database systems [52, 23, 63] which, essentially, extend relational database systems [95] by recursion and stratified negation. Then, in section 3.3 we analyze the more recent answer set programming systems [83, 36] which go beyond deductive databases, supporting also unstratified negation and other advanced constructs like disjunction, and various forms of constraints. 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). This chapter provides a comparative analysis of such systems. However, given the wide range of contexts possibly benefiting of their application, it is difficult to perform an objective comparison between them without fixing a context. We have chosen to characterize the presented systems when applied to the information integration problem. This because such a context has been the main test bed for our work.

3.1 Declarative computational logic 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 [51, 50, 67, 130, 94]. 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 [129]. 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* [52, 23, 63]. 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 [132, 23].

In the past few years, various set-oriented evaluation strategies specific for deductive databases have been the main focus of extensive research [12, 13, 16, 23, 66, 70, 98, 118, 132, 131] and a number of deductive database systems or prototypes based on Datalog have been reported. These include Nail [97], LOLA [49], Glue-Nail [33], XSB [120], CORAL [112], Aditi [133], LogicBase [65], Declare/SDS [72], EKS(-V1) [138] etc. See [113] 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) [134, 135, 136, 75], integrity checking (Soundcheck) [29], the deductive database system EKS(-V1) [138], hypothetical reasoning and integrity constraints checking [137]. The EKS system used a top-down evaluation method.

Efforts at MCC emphasized bottom-up evaluation methods [129] and query evaluation using methods such as seminaive evaluation, magic sets and counting [12, 16, 117, 119], semantics for stratified negation and set-grouping [15], 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) [33]. The results of this work led to the first paper on recursion using the Magic Sets method [12], done in collaboration with the group at MCC. Other contributions were aggregation in logical rules, and theoretical contributions to negation: stratified negation [58], well founded negation [59] and modularly stratified negation [116]. A language, called GLUE [105, 96] 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 [112]. 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 [111]. 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 [12, 16].

3.1.1 Relevant features of computational logic systems

As outlined in the previous sections, deductive database systems have been used in a variety of application domains. This wide range of possible applications 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. The choice is also motivated by the fact that information integration has been the main real-word benchmark applications of the work developed in this thesis.

An important aspect of the data integration field consists in answering queries involving several data sources rather than one single database. As an example, the set of information 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 [20, 77, 76]. 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.

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.

3.1.2 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.
- *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 .

3.1.3 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 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;

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 3.1: Template table for describing systems' features

- complex queries (i.e., whose complexity goes beyond polynomial-time) over medium-sized data.

3.2 Deductive database systems

In the sequel of the chapter, we give an overview of the most important computational logic systems proposed in the literature and, for each of them, we analyze the features outlined in the previous section. In particular, we summarize these features in a table as shown in Table3.1 and describe, for each of them, the characteristics of the system.

3.2.1 LDL++

The LDL project [25] 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-based 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 3.2 we summarize the features of the LDL system w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

3.2.2 NAIL!

The NAIL! (Not Another Implementation of Logic) [33] project at Stanford University aims at supporting the optimal execution of Datalog goals over a relational database system. 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,

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 3.2: Features of LDL

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;
- *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 3.3 we summarize the features of the NAIL! system w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

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 3.3: Features of NAIL!

3.2.3 POSTGRES

POSTGRES [108] 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.

In Table 3.4 we summarize the features of POSTGRES w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

3.3 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[1] (ASP) is a logic programming alternative to SAT-based programming, which is

[1]Name coined by Vladimir Lifschitz in the invited talk at ICLP'99.

POSTGRES	
Language expressiveness	
Ability to express views	Yes
Ability to express recursive queries	No
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 3.4: Features of POSTGRES

successful and widely used in the area of Artificial Intelligence [71]. 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[2] correspond to solutions; an ASP system is then used to find such solutions of P [83].

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 $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$ [39]). Importantly, the ASP encoding of a large variety of problems is often very concise, simple, and elegant [36].

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 popular systems DLV and SMODELS.

3.3.1 DLV

The development of the DLV system (datalog plus vel, i.e., disjunction) [18, 27, 30] 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

[2]An answer set is a preferred model of an ASP program.

Austria.

The system is based on disjunctive logic programming without function symbols under the consistent answer set semantics [60] 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 [30, 41], 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 3.1. 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 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) [42, 43], 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.

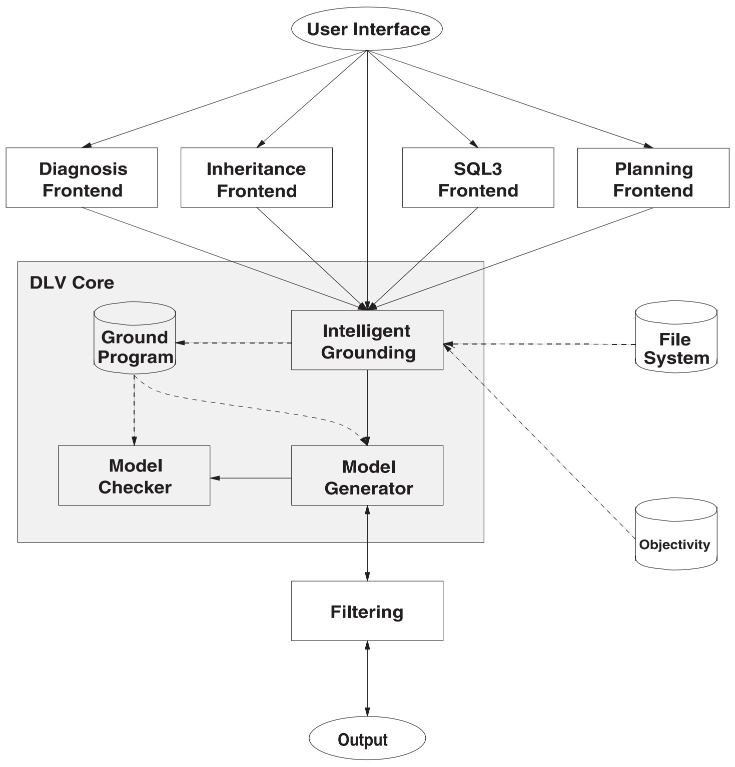


Figure 3.1: The System Architecture of DLV

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 3.5 we summarize the features of the DLV system w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

3.3.2 SMODELS

The SMODELS system [101, 100] 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 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

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 3.5: Features of DLV

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 [123].

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.

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 [69]. 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 3.6 we summarize the features of SMODELS w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

3.3.3 Cmodels

Cmodels is a system that computes answer sets for either disjunctive logic programs or logic programs containing choice rules. Answer set solver Cmodels uses SAT solvers as a search engine for enumerating models of the logic program – possible solutions, in case of disjunctive programs SAT solver zChaff is also used for verifying the minimality of found models.

The system Cmodels is based on the relation between two semantics: the answer set and the completion semantics for logic programs. For big class of programs called tight, the answer set semantics is equivalent to the completion semantics, so that the answer sets for such a program can be enumerated by a SAT solver.

On the other hand for nontight programs [86], and [74] introduced the concept of the loop formulas, and showed that models of completion extended by all the loop formulas of the program are equivalent to the answer sets of the program. Unfortunetly number of loop formulas might be large, therefore computing all of them may become computationally expensive. This led to the adoption of the algorithm that computes loop formulas "as needed" for finding answer sets of a program.

Cmodels is similar to Smodels or GnT in that its input is a grounded logic program that can be generated by the front-end called Lparse. The input of Cmodels may contain weight constraints, but optimize statements are not allowed. The representation of weight constraints by propositional formulas used in Cmodels is based on [47].

Table 3.7 summarizes the features of Cmodels relevant to data integration tasks.

3.3.4 ASSAT

ASSAT (Answer Sets by SAT solvers) [85] 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.
- 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 [85], 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 3.8 we summarize the features of ASSAT w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

3.3.5 noMoRe

The **no**n-**mo**notonic **re**asoning system `noMoRe` [4] 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 [87, 88] 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 [88]. 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 [3] and it was also successfully tested with `SWI` -Prolog [140].

`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` [99] 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 3.9 we summarize the features of `noMoRe` w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

3.3.6 SLG

The SLG system [24] 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.

The SLG-$\forall$ 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-$\forall$ computes the alternating fixpoint semantics by processing universal rules directly.

A profiling of the SLG-$\forall$ 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 3.10 we summarize the features of SLG w.r.t. the properties relevant to data integration we have pointed out in Section 3.1.1.

3.3.7 DeReS

The system DeReS [26] 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 [26] 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.

The features of the system DeReS are summarized in Table 3.11.

3.3.8 XSB

The XSB system is an inmemory deductive database engine based on a Prolog/SLD resolution strategy. Clearly, the traditional Prolog systems are known to have serious deficiencies when used as database systems. Indeed, the SLD computational mechanism, which well serves the needs of a programming language, is clearly inadequate as a database computation strategy. Its most serious drawback is that it does not terminate for the datalog language. Datalog is a decidable language (one reason that makes it a reasonable candidate for a database language) but SLD refutation is not finite on it. The deductive database community has adopted datalog as a leading database query language, identified these problems, and rectified them. Rewriting techniques have been developed to introduce goaldirectedness into a bottomup, setatatime evaluation strategy. These techniques solve SLD's problems of lack of finiteness and redundant computation. XSB offers an alternative approach to creating a deductive database system. Rather than depending on rewriting techniques, it extends Prolog's SLD resolution in two ways: 1) adding tabling to make evaluations finite and nonredundant on datalog, and 2) adding a scheduling strategy and delay mechanisms to treat general negation efficiently. The resulting strategy is called SLG resolution, which is complete and finite for nonfloundering programs with finite models, whether they are stratified or not.

The system XSB [114] 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. It is worth pointing out that XSB can work only in main memory and, consequently, it could not evaluate programs working on huge amounts of data.

Table 3.12 summarizes the features of XSB relevant to data integration tasks.

3.3.9 claspD

`claspD` is an answer set programming (ASP) solver for (extended) normal and disjunctive logic programs. It is able to deal with problems at the second level of the polynomial hierarchy. `claspD` deploys a generate and test approach, both tasks implemented by way of clasp's core technology [54]; consequently, it combines the high-level modeling capacities of Answer Set Programming with state-of-the-art techniques from the area of Boolean constraint solving.

Unlike existing ASP solvers, claspD is originally designed and optimized for conflict-driven ASP solving [56, 55], centered around the concept of a nogood from the area of constraint processing (CSP). Rather than applying a SAT(isfiability checking) solver to a CNF conversion, clasp directly incorporates suitable data structures, particu- larly fitting backjumping and learning.

Such techniques include:

- conflict analysis via the First-UIP scheme;
- nogood recording and deletion;
- backjumping;
- restarts;
- conflict-driven decision heuristics;
- unit propagation via watched literals;
- dedicated propagation of binary and ternary nogoods;

However, claspD is a genuine ASP solver. Its basic propagation engine includes advanced unfounded set checking based on source pointers. In fact, claspD is the first disjunctive ASP solver whose propagation engine is able (but not guaranteed) to detect non-singleton unfounded sets within non-head-cycle-free strongly connected components of a program's (positive) atom dependency graph. Furthermore, non-polynomial unfounded set checks are only applied if necessary, that is, when an exhaustive test of a non-head-cycle-free strongly connected component is needed.

Table 3.13 summarizes the features of claspD relevant to data integration tasks.

3.3.10 Other systems

Among other ASP systems we cite the system *near Horn* [89, 92] which has been implemented in PROLOG; in [121] the system DisLog is described, which incorporates different disjunctive theories and strategies including the semantics introduced in [90]; DisLog tries to eliminate redundant computations by using a breadth-first approach. The system DisLoP [6, 7] 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 [35], which allows variables but not function symbols in the language. The GnT system is an implementation of the stable model semantics for disjunctive logic programs constructs on top of Smodels system; this implementation is based on an architecture consisting of two interacting Smodels solvers for non-disjunctive programs, one of the them is responsible for generating as good as possible model candidates while the other checks for minimality, as required from disjuctive stable 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 3.6: Features of SMODELS

Cmodels	
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 and disjunction
Ability to deal with computationally hard queries	Up to Σ_2^P
Other	
Efficiency	
Efficiency with simple but data intensive queries	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 3.7: Features of Cmodels

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 3.8: Features of ASSAT

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 3.9: Features of noMoRe

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 3.10: Features of SLG

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 3.11: 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 3.12: Features of XSB

claspD	
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 and disjunction
Ability to deal with computationally hard queries	Up to Σ_2^P
Other	
Efficiency	
Efficiency with simple but data intensive queries	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 3.13: Features of claspD

Chapter 4

DLV^{DB} - Main Features

Disjunctive logic programming under answer set semantics (DLP, ASP) is a powerful formalism for knowledge representation and reasoning. The language of DLP is very expressive, and allows for modelling complex combinatorial problems. However, despite the high expressiveness of this language, the success of DLP systems is still dimmed when the applications of interest become data intensive (current DLP systems work only in main memory) or they involve some inherently procedural sub-tasks or the handling of complex data structures.

DLV^{DB} is conceived as an extension of the DLV system allowing the instantiation of logic programs directly on databases and to handle input and output data distributed on several databases in order to combining the expressive power of DLP with the efficient data management features of DBMS*s*.

Part of the material presented in this chapther appeared in [147, 151, 148, 149, 152, 150].

4.1 Introduction

The main limitations of current DLP systems in real world scenarios can be summarized in four main issues : *(i)* they are not capable of handling data intensive applications (they work in main memory only), *(ii)* they provide a limited interoperability with generic, external, DBMSs, *(iii)* they are not well suited for modelling inherently procedural problems, and *(iv)* they can not reason about recursive data structures and infinite domains, such as XML/HTML documents.

DLV^{DB} has been conceived to increase the cooperation between ASP systems and databases; it allows substantial improvements in both the evaluation of logic programs and the management of input and output data distributed on several databases.

Moreover, DLV^{DB} presents enhanced features to improve its efficiency and usability in the contexts outlined above, for an effective exploitation of DLP in real world scenarios. These features include:

- Full support to disjunctive datalog with unstratified negation, and aggregate functions;
- Extension of DLP with external function calls, particularly suited for solving inherently procedural sub-tasks but also for improving knowledge-modelling power;
- Extension of DLP to support list terms;

- An evaluation strategy devoted to carry out as much as possible of the reasoning tasks in mass memory, thus enabling complex reasonings in data intensive applications without degrading performances.
- Primitives to integrate data from different databases, in order to easily specify which data is to be considered as input or as output for the program.

In order to make the above features possible, various challenges had to be faced:

1. Data intensive applications usually must access, and modify, data stored in autonomous enterprise databases and these should be accessed also by other applications.
2. Evaluating the stable models of an ASP program directly in mass-memory data-structures, could be highly inefficient.
3. Using the main memory to accommodate both the input data (hereafter, EDB) and the inferred data is usually impossible for data intensive applications due to the limited amount of available main memory.
4. The introduction of functions and list terms makes the evaluation of programs more complex.

In order to face challenge 1. DLV^{DB} is interfaced with external databases via ODBC. ODBC allows a very straightforward way to access and manipulate data over, possibly distributed, databases. Moreover, in order to properly carry out the evaluation of a program, it is necessary to specify the mappings between input and output data and program predicates, as well as to specify wether the temporary relations possibly needed for the mass-memory evaluation should be maintained or deleted at the end of the execution. These can be specified by some *Auxiliary Directives*, more details on these directives are discussed in Section4.1. Note that challenge 1. makes the adoption of deductive systems integrating proprietary DBMSs not effective.

As far as challenge 2. is concerned we adopt a mixed strategy, a detailed description of the evaluation strategies adopted by DLV^{DB} is presented in Section 5.2; intuitively, the evaluation can be divided in two distinct phases: the grounding and the model generation. Grounding is completely performed in the database, whereas the model generation is carried out in main memory; this allows also to address challenge 3. In fact, in several cases, only a small portion of the ground program is actually needed for the model generation phase, since most of the inferred data is “stable” and belongs to every stable model (and is already derived during the grounding phase).

Note that, from points 2. and 3. it comes out that some amount of data *must* be loaded in main memory, but this should be as small as possible.

Finally, as for challenge 4. we exploit database stored functions to implement external function calls; these are also the basis for supporting list terms, which are handled with suitable manipulation functions. Functions are introduced in the program by estending the supported language with external atoms, details on the corresponding syntax have beeen presented in Section 2.1, whereas, evaluation of programs with functions and list terms is shown in Section5.4.

In the following we focus on the Auxiliary Directives the user can specify to let DLV^{DB} interact with external databases.

4.2 Auxiliary directives

DLV^{DB} allows the user to specify a variety of auxiliary directives for handling the interaction between the system and a set of (possibly distributed) databases. The grammar in which these directives must be expressed is shown in Figure 4.1.

```
Auxiliary-Directives ::=
   Init-section
   [Table-definition]+
   [Function-definition]*
   [Query-Section]?
   [Final-section]*

Init-Section ::=
   USEDB DatabaseName:UserName:Password [System-Like]?.

Table-definition ::=
   [USE TableName [( AttrName [, AttrName]* )]?
   [AS ( SQL-Statement )]?
   [FROM DatabaseName:UserName:Password]?
   [MAPTO PredName [( SqlType [, SqlType]* )]? ]?.
   |
   CREATE [VIEW]? TableName [( AttrName [, AttrName]* )]?
   [MAPTO PredName [( SqlType [, SqlType]* )]? ]?
   [KEEP_AFTER_EXECUTION]?.]

Function-definition ::=
   USEFUNCTION FunctionName MAPTO ExternalPredName.

Query-Section ::= QUERY TableName.

Final-section ::=
   [DBOUTPUT DatabaseName:UserName:Password.
   |
   OUTPUT [APPEND | OVERWRITE]? PredName [AS AliasName]?
   [IN DatabaseName:UserName:Password.]

System-Like ::=
   LIKE [POSTGRES | ORACLE | DB2 | SQLSERVER | MYSQL]
```

Figure 4.1: Grammar of the auxiliary directives.

The auxiliary directives can be subdivided in five sections, namely the *init section*, the *table definition* section, the *function definition* section, the *query section* and the *final section*.

The *init section* defines the working database on which the instantiation process will be carried out; this database, like all the other ones, is accessed via an ODBC connection that must be previously set up and, consequently, username and password for the connection must be provided here. The `System-Like` option can be used here to specify the kind of DBMS which implements the working database, in order to exploit tha corresponding SQL dialect. If this option is omitted, the working DBMS is assumed to be a generic DBMS supporting standard SQL.

The *table definition* section specifies the mappings between logic program predicates and database tables (or SQL views). The user can exploit two options in this phase, namely the `USE` and the `CREATE` option.

The `USE` option must be exploited if the table `TableName` already exists and the data it contains must be exploited as input (facts) for the instantiation process. Note that this data might reside on a database different from the working database (this situation can be specified with the option `[FROM DatabaseName]`). If the data is available, but it is not in a single table, `TableName` can be filled with the result of the SQL query specified in the option `[AS (SQL-Statement)]`.

The `CREATE` option must be exploited when the table must be created (if a table with the same name is present in the database it is first removed and then redefined) in the working database from the output computed by the program. If the user specifies the option `KEEP_AFTER_EXECUTION`, the table is kept in the working database at the end of the instantiation process; otherwise it is removed.

Of particular interest is the `VIEW` option of the `CREATE` statement (not available in the earlier version of DLV^{DB}); this can be used to specify that tables associated with intermediate predicates (i.e. corresponding to neither input nor output data) should be maintained virtual. This possibility may provide both space saving and performance improvements, especially for those programs having a hierarchical structure. Note that `VIEW` option cannot be used with both recursive and unsolved predicates[1].

In both cases, the `MAPTO` option must be exploited to link the table to a predicate. The number of attributes of the table must be equal to the arity of the associated predicate. In the definition of tables, attribute type declaration is needed only if the program contains built-in predicates or aggregate functions for a correct management of them.

If a predicate is not explicitly mapped into a table, but a relation with the same name and compatible attributes is present in the working database, the system automatically creates a `USE` mapping for them. Analogously, if a predicate is not explicitly mapped and no corresponding table exists in the working database, a `CREATE` mapping is automatically generated for it. These functionalities significantly simplify the specification of the auxiliary directives (in the ideal case – when everything is in the working database and all the input predicates have the corresponding input table having the same name – only the init section and one of `CREATE` or `OUTPUT` options are actually needed to run a program and check its output).

The *function definition* section specifies the mappings between logic program external predicates and database stored functions.

The *query section* can be used to specify the database table devoted to store the results of the query possibly present in the program.

The *final section* allows to copy either the entire output of the instantiation or only some of the tables on a database different from the working database. If the user does not specify any table name, the entire output is copied; otherwise, only the specified tables are copied. In the latter situation, the user can choose to append the data created during the instantiation to those already present in the target table (by the option `APPEND`) or to overwrite the (possibly existing) data in the target table (by the option `OVERWRITE`); if no option is specified, `OVERWRITE` is the default. The definition of the target table is the same as the source one except for the name which can be changed to `AliasName`. Note that directives specified in the *final section* must be used

[1]For a definition of *unsolved predicate* we refer to Section 5.2.

only with option `-n=1`, this option causes the system to compute one stable model only.

These directives provide DLV^{DB} with high flexibility in source data location and output management.

Example 4.2.1 Assume that a travel agency asks to derive all the destinations reachable by an airline company either by using its vectors or by exploiting code-share agreements. Suppose that the direct flights of each company are stored in a relation `flight_rel(Id, From, To, Company)` of the database `dbAirports`, whereas the code-share agreements between companies are stored in a relation `codeshare_rel (Company1, Company2, FlightId)` of an external database `dbCommercial`; if a code-share agreement holds between the company $c1$ and the company $c2$ for $flightId$, it means that the flight $flightId$ is actually provided by a vector of $c1$ but can be considered also carried out by $c2$. Finally, assume that, for security reasons, it is not allowed to travel agencies to directly access the databases `dbAirports` and `dbCommercial`, and, consequently, it is necessary to store the output result in a relation `composedCompanyRoutes` of a separate database `dbTravelAgency` supposed to support travel agencies. The DLP$^{\mathcal{A}}$ program that can derive all the connections is:

$$\begin{array}{lll} (1) & destinations(From, To, Comp) & \text{:-} \quad flight(Id, From, To, Comp). \\ (2) & destinations(From, To, Comp) & \text{:-} \quad flight(Id, From, To, C2), \\ & & \phantom{\text{:-}} \quad codeshare(C2, Comp, Id). \\ (3) & destinations(From, To, Comp) & \text{:-} \quad destinations(From, T2, Comp), \\ & & \phantom{\text{:-}} \quad destinations(T2, To, Comp). \end{array}$$

In order to exploit data residing in the above mentioned databases, we should map the predicate $flight$ with the relation `flight_rel` of `dbAirports` and the predicate $codeshare$ with the relation `codeshare_rel` of `dbCommercial`. Finally, we have to map the predicate $destinations$ with the relation `composedCompanyRoutes` of `dbTravelAgency`.

Now suppose that, due to a huge size of input data, we need to perform the program execution in mass-memory (on a DBMS). In order to carry out this task, the auxiliary directives shown in Figure 4.2 should be used. They allow to specify the mappings between the program predicates and the database relations introduced previously.

```
USEDB dlvdb:myname:mypasswd.

USE flight_rel (Id, From, To, Company)
FROM dbAirports:airportUser:airportPasswd
MAPTO flight (integer, varchar(255), varchar(255), varchar(255)).

USE codeshare_rel (Company1, Company2, FlightId)
FROM dbCommercial:commUser:commPasswd
MAPTO codeshare (varchar(255), varchar(255), integer).

CREATE destinations_rel (From, To, Company)
MAPTO destinations (varchar(255), varchar(255), varchar(255))
KEEP_AFTER_EXECUTION.

OUTPUT  destinations AS composedCompanyRoutes
IN dbTravelAgency:agencyName:agencyPasswd.
```

Figure 4.2: Auxiliary directives for Example 4.2.1.

Chapter 5

DLV^{DB} - Implementation Principles

This chapter provides the description of implementation principles of the DLV^{DB} system. First we describe the architecture of DLV^{DB}, evidencing the strong relation with the DLV architecture. Then, we present the evaluation strategy implemented in DLV^{DB} devoted to carry out as much as possible of the reasoning tasks in mass memory without degrading performances. Moreover, we present a fixpoint computation technique (Differential Semi-Naive Evaluation) that reduces the redundancy in the inference process and functions for translating DLP rules into SQL statements.

Part of the material presented in this chapther appeared in [147, 151, 148, 149, 152, 150].

5.1 System Architecture

In this section we present the general architecture of the system. It has been designed as an extension of the DLV system [80], which allows both instantiating of logic programs directly on databases and the handling of input and output data distributed on several databases. It combines the expressive power of DLV (and the optimization strategies implemented in it) with the efficient data management features of DBMSs [53].

Figure 5.1 illustrates the architecture of the system; in the figure, the boxes marked with DLV are the ones already developed in the DLV system. An input program $\mathcal{P}$ is first analyzed by the Parser which encodes the rules in the intensional database (IDB) in a suitable way and builds an extensional database (EDB) in main-memory data structures from the facts specified directly in the program (if any). As for facts already stored in database relations, no EDB is produced in main-memory. After this, the Optimizer applies a rewriting procedure in order to get a program $\mathcal{P}'$, equivalent to $\mathcal{P}$, that can be instantiated more efficiently and that can lead to a smaller ground program. The Dependency Graph Builder computes the dependency graph of $\mathcal{P}'$, its connected components and a topological ordering of these components. Finally, the DB Instantiator module, the core of the system, is activated. The DB Instantiator module receives:

1. the IDB and the EDB (if not empty) generated by the parser;
2. the Dependency Graph (DG) generated by the dependency graph builder;
3. the auxiliary directives specifying the needed interactions between DLV^{DB} and the databases.

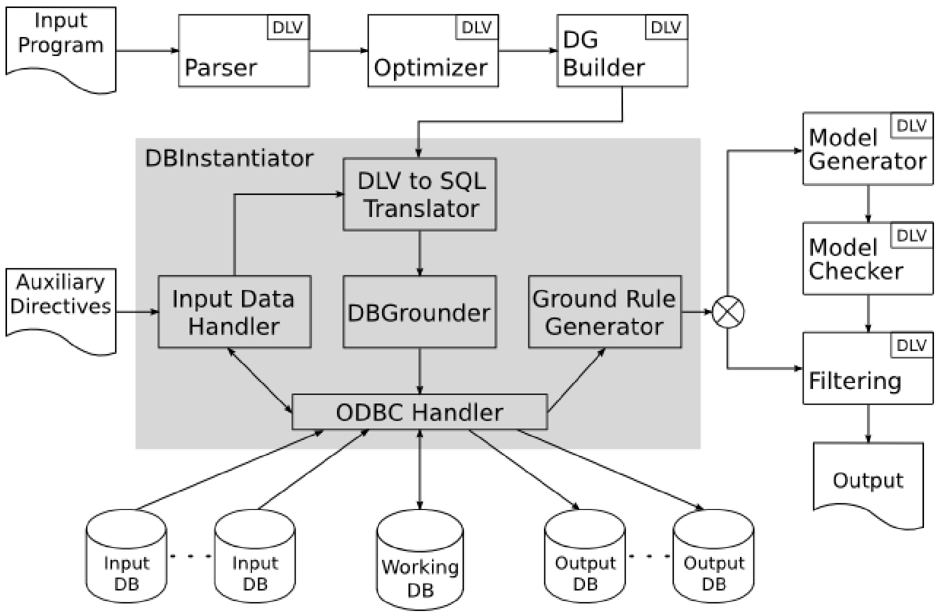


Figure 5.1: Architecture of DLV^{DB}.

First, each DLV rule is translated into SQL statement by the *DLV to SQL Translator*, then these statements are executed on the *Working Database* by the *DBGrounder*. The *Input Data Handler* receives the auxiliary directives defined by the user and performs all mappings and settings needed for a correct translation and execution of the SQL statements produced by the *DLV to SQL Translator*.

Finally, the *Ground Rule Generator* extracts data from database, create ground rules (if any) and load them to *Model Generator*. The *Model Generator* produces some “candidate” answer sets (models), the stability of which are verified by the *Model Checker*; then each stable model is printed on output.

Note that, if no ground rule is generated after grounding, it means that all the program has been solved by the grounder, then the *Ground Rule Generator* prints directly on output the (unique) stable model found.

It is worth pointing out that all the instantiation steps are performed directly on the working database through the execution of SQL statements and no data is loaded in main-memory from the databases in any phase of the grounding process.

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

It is important to point out that the architecture of DLV^{DB} has been designed in such a way to fully exploit optimizations both from logic theory and from database theory. In fact, the actually evaluated program is the one resulting from the Optimizer module which applies program rewriting techniques aiming at simplifying the evaluation process and reducing the dimension of the ground program. Then, the execution of the SQL statements in the query plan exploits data-oriented optimizations implemented in the DBMS. As far as this latter point is concerned, we have experienced that the typology of the working database associated with DLV^{DB} may sensibly affect system performance; in fact, when DLV^{DB} was coupled with highly sophisticated DBMSs it generally showed better performance in handling great amounts of data w.r.t. the same executions when coupled with less sophisticated DBMSs.

5.2 Evaluation Strategy

The main innovation of the proposed system resides in the instantiation of DLP programs directly on a database. The instantiation process basically consists of two steps: *(i)* the translation of DLP rules in SQL statements, *(ii)* the definition of an efficient query plan, which basically consists in an enhancement of the Semi-Naive evaluation.

The evaluation strategy implemented in our approach puts its basis on the sharp distinction existing between the grounding of the input datalog program and the generation of its stable models. Then, two distinct approaches can be adopted whether the input program is non disjunctive and stratified (in this case everything can be evaluated on the DBMS) or not.

5.2.1 Evaluation of non disjunctive stratified programs

It is well known that if a program is non disjunctive and stratified, it has a unique stable model corresponding exactly to its ground instantiation. The evaluation of these kinds of program, intuitively, consists in the translation of each datalog rule in a corresponding SQL statement and in the composition of a suitable query plan on the DBMS; the evaluation of recursive rules is carried out with an improved semi-naïve approach.

Before starting the evaluation of a (possibly optimized) program $\mathcal{P}$, its connected components and their topological order (i.e., the Dependency Graph of $\mathcal{P}$) are computed. Then, the program is evaluated one component at a time, starting from the lowest ones in the topological order. This process is iterated until no new ground instance can be derived from the component rules.

At each iteration, the instantiation of a rule consists of the execution of the SQL statement associated with it. One of the main objectives in the implementation of DLV^{DB} has been that of associating one single (non recursive) SQL statement with each rule of the program (either recursive or not), without the support of main-memory data structures for the instantiation. This allows DLV^{DB} to fully exploit optimization mechanisms implemented in the DBMSs and to minimize the “out of memory” problems caused by limited main-memory dimensions.

Since the handling of recursive rules and the translation of datalog to SQL are problems addressed also in the evaluatio of disjunctive programs with unstratified negation we analyze them later in Secton 5.3 and 5.4.

5.2.2 Evaluation of disjunctive programs with unstratified negation

In presence of disjunctive rules or unstratified negation in a program $\mathcal{P}$, the ground instantiation of $\mathcal{P}$ is not sufficient to compute its stable models. Then, grounding and model generation phases must both be carried out.

Also in this case, the evaluation strategy we adopt carries out the grounding completely in the database, by the execution of suitable SQL queries. This phase generates two kinds of data: ground atoms (facts) valid in every stable model (and thus not requiring further elaboration in the model generation phase) and ground rules, summarizing possible values for a predicate and the conditions under which these can be inferred.

Facts compose the so called *solved* part of the program, whereas ground rules form the *residual program*, not completely solved by the grounding. As previously pointed out, one of the main challenges in our work is to load the smallest amount of information as possible in main memory; consequently, the residual program generated by the system should be as small as possible.

Model generation is then carried out in main memory with the technique described in [80].

Definition 5.2.1 Let p be a predicate of a program $\mathcal{P}$, p is said to be *unsolved* if at least one of the following holds: *(i)* it is in the head of a disjunctive rule; *(ii)* it is the head of at least one rule involved in unstratified negation; *(iii)* the body of a rule having p as head contains at least one unsolved predicate. p is said to be *solved* otherwise.

In our evaluation strategy, a ground predicate is associated with facts only in the ground program and, thus, with *certainly-true* values, i.e. values true in every stable model. On the contrary, an unsolved predicate p may be defined by both facts (certainly-true values) and ground rules; the latter identify *possibly-true* values for p, i.e. the domain of values p may assume in stable models.

Given an unsolved predicate p we indicate the set of its certainly-true values as p^s and the set of its possibly-true values as p^u.

Example 5.2.2 Consider the simple program:

$$\begin{array}{l} q(1,2). \\ p(3). \\ p(X) \vee p(Y) \text{ :- } q(X,Y). \end{array}$$

Here q is a solved predicate, whereas p is an unsolved predicate; in particular, $q(1,2)$ is a certainly-true value for q, $p(3)$ is a certainly-true value for p, whereas $p(1)$ and $p(2)$ are possibly-true values of p. Then, $p^s = \{3\}$, whereas $p^u = \{1,2\}$.

As previously pointed out, rules having an unsolved predicate may generate ground rules in the instantiation. Since we are interested in generating the smallest residual program as possible, ground rules are "epurated" of certainly-true values.

Definition 5.2.3 A *simplified ground rule* (g-rule in the following) of a program $\mathcal{P}$ is a ground rule not involving any certainly-true values of $\mathcal{P}$.

Example 5.2.4 From the previous example, the (only) ground rule that can be generated is $p(1) \vee p(2)$:- $q(1,2)$. However this is not a simplified rule since it involves $q(1,2)$ which is a certainly-true value. Then, the corresponding g-rule is simply $p(1) \vee p(2)$.

It is now possible to illustrate the evaluation strategy implemented in our system. Consider a program $\mathcal{P}$ composed of non ground rules of the form:

$$\alpha_1 \vee \cdots \vee \alpha_k \text{ :- } \beta_1, \ldots, \beta_n, \text{not}\,\beta_{n+1}, \ldots, \text{not}\,\beta_m, \gamma_1, \ldots, \gamma_p, \text{not}\,\gamma_{p+1}, \ldots, \text{not}\,\gamma_q. \quad (5.1)$$

where β_i (resp. γ_j) are solved (resp. unsolved) predicates. The evaluation is carried out in five steps:

Step 1. Translate $\mathcal{P}$ in an equivalent program $\mathcal{P}'$;

Step 2. Translate each rule of $\mathcal{P}'$ in a corresponding SQL statement;

Step 3. Compose and execute the query plan of statements generated in Step 2 on the DBMS;

Step 4. Generate the residual program and load it in the Model Generator of DLV;

Step 5. Execute the residual program in main memory and show the results.

Step 1. The objective of this step is to "prepare" rules of $\mathcal{P}$ to be translated in SQL almost straightforwardly, in order to generate a residual programs as small as possible. In more detail, for each rule r in $\mathcal{P}$ three kinds of rule are generated:

A. If the head of r has one atom only ($k = 1$), a rule (hereafter denoted as A-rule) is created for deriving only certainly-true values of r's head; note that if $k > 1$ no certainly-true values can be derived from r.

B. A set of rules (hereafter, B-rules) supporting the generation of the g-rules of r. The heads of these rules contain both the variables of unsolved predicates in the body of r and the variables in the head of r. Ground values obtained for these variables with B-rules are then used to instantiate r with possibly-true values only.

C. A set of rules (hereafter, C-rules) for generating the set of possibly-true values of unsolved predicates as projections on B-rules obtained previously.

Given a generic rule defined as (5.1), the corresponding A-rule have the form:

$$\alpha_1^s \text{ :- } \beta_1, .., \beta_n, \text{not}\, \beta_{n+1}, .., \text{not}\, \beta_m, \gamma_1^s, .., \gamma_p^s, \text{not}\, \gamma_{p+1}^s, \text{not}\, \gamma_{p+1}^u, .., \text{not}\, \gamma_q^s, \text{not}\, \gamma_q^u. \qquad (5.2)$$

where for positive unsolved predicates only certainly-true values ($\gamma_1^s, \ldots, \gamma_p^s$) are considered, whereas for negated unsolved predicates both certainly-true and possibly-true values ($\gamma_{p+1}^s, \ldots, \gamma_q^s$, $\gamma_{p+1}^u, \ldots, \gamma_q^u$) must be taken into account.

Example 5.2.5 Consider the following program, which will be exploited as a running example throughout the rest of the section:

$$\begin{array}{ll} r1: & q(1,2). \\ r2: & p(Y,X) \vee t(X) \text{ :- } q(X,Y). \\ r3: & q(X,Y) \text{ :- } p(X,Y), not\, t(X). \end{array}$$

Here both p, q, and t are unsolved. The A-rules derived for this program are:

$$\begin{array}{ll} r1.A: & q^s(1,2). \\ r3.A: & q^s(X,Y) \text{ :- } p^s(X,Y), not\, t^s(X), not\, t^u(X). \end{array}$$

where rule $r2$ does not contribute since it is disjunctive and cannot generate certainly-true values.

B-rules play a key role in our approach; in fact, they allow the generation of the residual program. In particular, their role is to identify the set of values for variables in unsolved predicates of the body of r, generating possibly-true values of the head of r. Then, r is seen as a template for generating its g-rules, and ground values derived by the corresponding B-rules are used to instantiate r.

Note that in order to generate a possibly true value for a normal rule, at least one possibly true value must be involved in its body, whereas disjunctive rules always generate possibly-true values. Moreover, in order to properly generate g-rules (i.e. ground rules involving possibly-true values only) the system must be able to track, for each truth value of a B-rule, which predicates of r contributed with a certainly-true value and which ones with a possibly-true value.

In our approach, this issue is addressed by first labelling each unsolved predicate γ_j of r alternatively with a 0 or with a 1, where a 0 indicates to take its γ_j^s, whereas a 1 indicates to consider its γ_j^u. Then, each binary number between 1 and 2^q-1 for normal rules and between 0 and 2^q-1 for disjunctive rules[1] corresponds to a labelling stating the combination of values to be considered. For each labelling, a corresponding B-rule is generated starting from the definition of r and substituting each unsolved predicate γ_j with γ_j^s (resp., γ_j^u) if the corresponding label is 0 (resp., 1).

The only exception is caused by negated unsolved predicates. In fact, if γ_j is negated and labelled with a 1, it must be put in the B-rule without negation. In fact, negated certainly-true values surely invalidate the satisfiability of the g-rule, whereas negated possibly-true values may invalidate the rule only if the model generator sets them to true.

It is worth pointing out that our labelling approach, makes significantly easier the generation of simplified ground rules; in fact, it is sufficient to consider only the values of predicates labelled with 1 and not derived to be certainly-true by other rules.

Finally, in order to allow a proper reconstruction of g-rules from B-rules, a mapping between the variables of the B-rules and the variables of r is maintained.

Example 5.2.6 From rules $r2$ and $r3$ introduced in the previous example, the following B-rules are derived. Original rules are re-proposed in parenthesis to simplify the comprehension; labels are reported in rule names. Variable mapping is trivial and not reported.

$$\begin{array}{rl}
 & (r2 : p(Y,X) \vee t(X) \text{ :- } q(X,Y).) \\
r2.B(0): & B\text{-}rule\_r2(X,Y) \text{ :- } q^s(X,Y). \\
r2.B(1): & B\text{-}rule\_r2(X,Y) \text{ :- } q^u(X,Y). \\
 & (r3 : q(X,Y) \text{ :- } p(X,Y), not\ t(X).) \\
r3.B(01): & B\text{-}rule\_r3(X,Y) \text{ :- } p^s(X,Y), t^u(X). \\
r3.B(10): & B\text{-}rule\_r3(X,Y) \text{ :- } p^u(X,Y), not\ t^s(X). \\
r3.B(11): & B\text{-}rule\_r3(X,Y) \text{ :- } p^u(X,Y), t^u(X).
\end{array}$$

Finally, C-rules are simple projections on the B-rule heads over the attributes of the corresponding predicate.

Example 5.2.7 From rules $r2$ and $r3$ introduced previously and from the corresponding B-rules the system generates:

[1] Recall that q is the number of unsolved predicates in the body of r.

$$\begin{array}{ll} r2.C\_p: & p^u(Y,X) \texttt{ :- } B\text{-}rule\_r2(X,Y). \\ r2.C\_t: & t^u(X) \texttt{ :- } B\text{-}rule\_r2(X,Y). \\ r3.C\_q: & q^u(X,Y) \texttt{ :- } B\text{-}rule\_r3(X,Y). \end{array}$$

Note that in the example above, the same B-rule predicate (B-rule_r2) is used to generate possibly-true values of two predicates (p^u and t^u); this follows directly from the fact that $r2$ is a disjunctive rule involving p and t.

Step 2. Translation of the rules obtained in Step 1. into SQL is carried out with the technique already presented in [152] for non disjunctive and stratified programs. As an example, rule $r3.A$ introduced above is translated into[2]:

INSERT INTO q^s (SELECT p^s.att$_1$, p^s.att$_2$, FROM p^s
    WHERE p^s.att$_1$ NOT IN (SELECT * FROM t^s)
    AND p^s.att$_1$ NOT IN (SELECT * FROM t^u))

Step 3. In order to compile the query plan, the dependency graph D associated with $\mathcal{P}$ is considered [81]. In particular, D allows the identification of a partially ordered set $\{\text{Comp}_i\}$ of program components where lower components must be evaluated first.

Then, given a component Comp and a rule r in Comp, if r is not recursive, then the corresponding portion of query plan is as follows[3]: *(1)* evaluate (if present) the A-rule associated with r; *(2)* evaluate each B-rule obtained from r; *(3)* for each predicate in the head of r evaluate the corresponding C-rule.

If r is recursive, the portion of query plan above must be included in a fix-point semi-naïve evaluation, as described in Section 5.3.

Step 4 and 5. The generation of the residual program requires the analysis of values derived by B-rules only. Then, for each rule r and each corresponding B-rule (say, r.B(L)), first predicates having label 0 in L are purged from r, then r is instantiated with values of r.B(L); during this phase a further check is carried out to verify if some predicate value has been derived as certainly-true by other rules. In this case the predicate is removed from the g-rule for that instance. The residual program is then loaded in main memory for the generation of stable models. Note that each answer set found on this residual program shall be enriched with certainly-true values determined during the grounding.

Example 5.2.8 The residual program generated for our running example is:

$$\begin{array}{l} p(2,1) \vee t(1). \\ p(1,2) \vee t(2) \texttt{ :- } q(2,1). \\ q(2,1) \texttt{ :- } p(2,1), not\ t(2). \end{array}$$

Note that the first g-rule does not involve q since it derives from r2.B(0), having $q(1,2)$ as certainly-true value.

[2]Here and in the following we use the notation x.att$_i$ to indicate the i-th attribute of the table x. Actual attribute names are determined at runtime.

[3]Here, for simplicity of exposition, we refer to rules, indicating that the corresponding SQL statements must be evaluated on the database.

5.3 Enhanced Semi-Naive method for evaluating recursive rules

In order to describe our approach, we first recall the classical Semi-Naive method; then we show some of its weaknesses and, finally, describe the improvement.

Given a program $\mathcal{P}$ and the associated dependency graph DG$_{\mathcal{P}}$ it is possible to single out an ordered sequence of components $\langle \mathcal{P}_{C_1}, \ldots, \mathcal{P}_{C_h} \rangle$ such that the evaluation of $\mathcal{P}_{C_g}$ $(1 \leq g \leq h)$ depends only on the evaluation of the components $\mathcal{P}_{C_f}$ such that $f < g$. Then, the Semi-Naive evaluation strategy considers one component of $\mathcal{P}$ at a time following the order specified by DG$_{\mathcal{P}}$.

The Semi-Naive algorithm applied to each component can be viewed as a two-phase algorithm: the first one deals with non-recursive rules, which can be completely evaluated in one single step; the second one deals with recursive rules which need an iterative fixpoint computation for their complete evaluation. At each iteration of the evaluation of a component $\mathcal{P}_{C_g}$, there are a number of predicates whose extensions have been already fully determined (predicates not belonging to $\mathcal{P}_{C_g}$ which have been therefore previously evaluated), and a number of recursive predicates (i.e., belonging to $\mathcal{P}_{C_g}$) for which a new set of truth values can be determined from the available ones. Let p_j be one of these recursive predicates, we indicate by Δp_j^k the set of new values determined for p_j at step k (in the following, we call Δp_j^k the *differential* of p_j).

Now, let p :- $\phi(p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_m)$. be a recursive rule such that ϕ is a first order formula, $p_1, p_2, \ldots p_n$ are mutually recursive to p, and $q_1, q_2, \ldots q_m$ are base or derived predicates which are non mutually recursive to p. The Semi-Naive method aims at reducing as much as possible the number of truth values computed at step k which were already true at step $k-1$. In order to achieve this goal it evaluates, at each iteration of the fixpoint computation, the following formula:

$$\begin{aligned}
&\Delta\phi(p_1^k, \Delta p_1^k, \ldots, p_n^k, \Delta p_n^k, q_1, \ldots, q_m) = \\
&\qquad \phi((p_1^k + \Delta p_1^k), (p_2^k + \Delta p_2^k), \ldots, (p_n^k + \Delta p_n^k), q_1, \ldots, q_m) - \\
&\qquad \phi(p_1^k, p_2^k, \ldots, p_n^k, q_1, q_2, \ldots, q_m).
\end{aligned}$$

where, with a little abuse of notation, we indicated with p_j^k the truth values of p_j computed until step k.

Thus, at each iteration, the Semi-Naive method evaluates only the differential of ϕ instead of the entire ϕ. However, as we will show with the next example, the Semi-Naive method still performs, at each iteration, some operations that can never contribute in computing new truth values and, as such, avoidable.

Consider the following rule in which g and h are themselves predicates mutually recursive to f:

$$f(X, Y) \text{ :- } f(X, Y), g(X, Y), h(X, Y).$$

In relational algebra, the evaluation of the ground instances of f is equivalent to:

$$F = F \bowtie G \bowtie H.$$

and, at step k, the standard Semi-Naive algorithm evaluates:

$$\begin{array}{lccccccl}
\Delta F^k = & \Delta F^{k-1} & \bowtie & G^{k-1} & \bowtie & H^{k-1} & \cup & (a) \\
 & F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-1} & \cup & (b) \\
 & F^{k-1} & \bowtie & G^{k-1} & \bowtie & \Delta H^{k-1} & & (c)
\end{array}$$

It is worth pointing out that the Semi-Naive evaluation strategy exploits three different tables for every recursive atom; in fact, for a generic relation T, it considers: T^{k-1}, i.e. the whole set of truth values computed for T until iteration $k-1$; ΔT^{k-1}, i.e. the new truth values computed for T at iteration $k-1$, and ΔT^k, i.e. the new values computed at iteration k from T^{k-1} and ΔT^{k-1}. Note that, in the Semi-Naive method, T^{k-1} contains also the tuples of ΔT^{k-1}. Clearly, each T^{k-1} can be explicitly split in T^{k-2} (i.e., the evaluation of T until step $k-2$), and ΔT^{k-1}; formally: $T^{k-1} = T^{k-2} \cup \Delta T^{k-1}$.

This observation allows to rewrite the formula for ΔF^k above in:

$$\Delta F^k = \begin{array}{c|ccccccl}
 & \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & H^{k-2} & \cup & \textit{(1)} \\
\textit{(a)} & \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & \textit{(2)} \\
 & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & \textit{(3)} \\
 & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(4)} \\
\hline
 & F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & \textit{(5)} \\
\textit{(b)} & F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(6)} \\
 & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & \textit{(7)} \\
 & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(8)} \\
\hline
 & F^{k-2} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & \textit{(9)} \\
\textit{(c)} & F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(10)} \\
 & \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & \textit{(11)} \\
 & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & . & \textit{(12)}
\end{array}$$

Now, observe that this new formula singles out some *join* operations that are *computed several times*, namely:

- join number (2) is equal to join number (11);
- join number (3) is equal to join number (7);
- join number (4) is equal to joins number (8) and (12);
- join number (6) is equal to join number (10);

This reasoning puts into evidence that, for the considered example, only 7 joins out of 12 (i.e., about 60%) are indeed needed to evaluate ΔF^k namely joins (1), (2), (3), (4), (5), (6), (9).

This situation can be generalized to any kind of recursive rule. As a consequence, a significant amount of avoidable operations, carried out at each step of the Semi-Naive evaluation, can be spared. Now, consider the following organization of the unavoidable joins above:

$$\begin{array}{ccccccl}
F^{k-2} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & \textit{(9)} \\
F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & \textit{(5)} \\
F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(6)} \\
\Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & H^{k-2} & \cup & \textit{(1)} \\
\Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & \textit{(2)} \\
\Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & \textit{(3)} \\
\Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(4)}
\end{array}
\left\{\begin{array}{ccc}
0 & 0 & 1 \\
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1
\end{array}\right.$$

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

Figure 5.2: Algorithm Differential Semi-Naive

if we associate the symbol 1 with a differential table and the symbol 0 with a standard table, it is possible to obtain the optimized join sequence that avoids the unnecessary operations in an automatic way. In particular, given a generic rule having r predicates in its body recursive with head, it is sufficient to suitably tag its standard and differential tables with the symbols 0 and 1 as shown previously and to follow a binary enumeration between 1 and $2^r - 1$.

We are now able to formalize the algorithm for a Differential Semi-Naive Evaluation strategy implemented in our system; it must be executed for each component $\mathcal{P}_{C_g}$ of the program $\mathcal{P}$; moreover, it assumes that input DLP rules have been already translated in SQL statements.

Let $\mathcal{P}_{C_g}$ be one component of a program $\mathcal{P}$ depending on predicates $r_1, \ldots, r_n$ solved in previous components and having $q_1, \ldots, q_m$ as non recursive predicates or facts and $p_1, \ldots, p_n$ as recursive predicates; let $R_1, \ldots, R_l$, (resp., $Q_1, \ldots, Q_m$ and $P_1, \ldots, P_n$) be the relations corresponding to $r_1, \ldots, r_l$ (resp., $q_1, \ldots, q_m$ and $p_1, \ldots, p_n$). Let ΔP_i^k be the differential relation storing only the new values computed for P_i at the current iteration k. Let ΔP_i^{k-1} be the differential relation storing the new values computed for P_i at iteration $k-1$ and let P_i^{k-2} be the content of the relation P_i at the end of iteration $k-2$. The Differential Semi-Naive Algorithm evaluates the minimal fixpoint for $\mathcal{P}_{C_g}$ as shown in Figure 5.2.

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

Function $EVAL\_DIFF(p_i, P_1^{k-2}, \ldots, P_n^{k-2}, \Delta P_1^{k-1}, \ldots, \Delta P_n^{k-1}, R_1, \ldots, R_l, Q_1, \ldots,\ Q_m)$ implements the optimization to the Semi-Naive method; it computes the new values for the predicate p_i at the current iteration k starting from the values computed until iteration $k-2$ and the new values obtained at the previous iteration $k-1$. In more detail, the SQL statements corresponding to each recursive rule having p_i as head are considered; as pointed out previously, in these statements standard and differential tables are exploited alternatively for the recursive predicates appearing in the body, whereas only standard tables are taken into account for non recursive predicates and facts. The final result of $EVAL\_DIFF$ is stored in table ΔP_i^k. Clearly, even if our approach performs less join operations than the Semi-Naive method it cannot be proved that $EVAL\_DIFF$ does not recompute some truth values already obtained in previous iterations. As a consequence, ΔP_i^k must be cleaned up from these values after the computation of $EVAL\_DIFF$; this is exactly what is done by instruction (5) of the algorithm.

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

5.4 From DLP to SQL

As previously pointed out, the evaluation strategy in DLV^{DB} is based on the translation of each DLP rule (recursive or not) in a single non recursive SQL statement. It is worth recalling that the system maps each predicate of the input program into a database relation, based on the specifications in the auxiliary directives and on the automatic mappings the system derives. In this section we provide the general functions exploited to perform the translations. Functions are presented in pseudocode and, for the sake of presentation clarity, they omit some details; moreover, since there is a one-to-one correspondence between the predicates in the logic program and the relations in the database, in the following, when this is not confusing, we use the terms predicate and relation interchangeably.

In order to provide examples for the presented functions, we exploit the following reference schema:

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

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

Translating Non-recursive Rules.

The function *TranslateNonRecursiveRule* is shown in Figure 5.3; it receives a rule r as input and returns the corresponding SQL Statement, depending on the rule typology. Here and in the following we use the operator $+$ to denote the "append" operator between strings. Function *head* receives the rule r and returns the relation associated with the atom in its head; this task is carried out by considering the mappings specified in the auxiliary directives. Function *isPositive(r)* (resp., *hasNegation(r)*, *hasBuilt-In(r)*, *hasNegationAndBuilt-In(r)*, *hasAggregate(r)*) receives a rule r and returns *true* if r is a positive rule (resp., contains negated atoms, contains built-in functions, contains both negated atoms and built-in functions, contains aggregate functions), *false* otherwise. In the following we describe in detail the corresponding translation functions[4].

Translating Positive Rules.

Intuitively, the SQL statement for positive rules is composed as follows: the SELECT part is determined by the variable bindings between the head and the body of the rule. The FROM part of the statement is determined by the predicates composing the body of the rule; variable

[4] *TranslateRuleWithNegationAndBuilt-In* will be not described since it is a straightforward fusion of *TranslateRuleWithNegation* and *TranslateRuleWithBuilt-In*.

```
Function **TranslateNonRecursiveRule**($r$: DLP$^{\mathcal{A}}$ rule): SQL statement
begin
    $SQL$:="";
    if (hasAggregate($r$)) then
        $SQL$:=$SQL$+TranslateAggregateRule($r$);
    $SQL$:=SQL+"INSERT INTO " + head($r$) + "(";
    if (isPositive(r)) then
        $SQL$:=$SQL$+TranslatePositiveRule($r$);
    else if (hasNegation(r)) then
        $SQL$:=$SQL$+TranslateRuleWithNegation($r$);
    else if (hasBuilt-In(r)) then
        $SQL$:=$SQL$+TranslateRuleWithBuilt-In($r$);
    else if (hasNegationAndBuilt-In(r)) then
        $SQL$:=$SQL$+TranslateRuleWithNegationAndBuilt-In($r$);
    $SQL$:=$SQL$+")";
    return $SQL$;
end.
```

Figure 5.3: Function TranslateNonRecursiveRule

bindings between body atoms and constants determine the WHERE conditions of the statement. Finally, an EXCEPT part is added in order to eliminate tuple duplications. The function is shown in Figure 5.4.

Here, function *head_attr(r)* returns the list of attributes of the relations in the body of r specified also in the head of r; the function returns this list in the proper order and also handles possible constant values specified in the rule head. Function *body$^+$(r)* returns the list of relations corresponding to the (positive) atoms present in the body of r. *joinConditions(r)* derives the join conditions to be specified among the involved relations from the positions and the names of the variables specified in the body of r, whereas *bodyConstantConditions(r)* handles possible constants specified in the body of r.

Example 5.4.1 Consider the following query:

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

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

```
INSERT INTO $q_0$ (
    SELECT  employee.$att_1$
    FROM employee,  department
    WHERE employee.$att_3$  =  department.$att_1$
        AND department.$att_2$='rossi'
        AND employee.$att_2$=100.000
    EXCEPT
        (SELECT * FROM $q_0$))
```

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

```
Function **TranslatePositiveRule**($r$: DLP$^{\mathcal{A}}$ rule): $SQL$ statement
begin
    $SQL$:=”SELECT ” + head_attr(r) +
        ”FROM ” + body$^{+}$($r$) +
        ”WHERE ” + joinConditions($r$) +
            ”AND ” + bodyConstantConditions($r$) +
        ”EXCEPT (SELECT * FROM ” + head($r$) + ”)”;
    return $SQL$;
end.
```

Figure 5.4: Function TranslatePositiveRule

```
Function **TranslateRuleWithNegation**($r$: DLP$^{\mathcal{A}}$ rule): $SQL$ statement
begin
    $SQL$:=”SELECT ” + head_attr(r) +
        ”FROM ” + body$^{+}$($r$) +
        ”WHERE ” + joinConditions($r$) +
            ”AND ” + bodyConstantConditions($r$);
    for each $p$ in body$^{-}$(r)
        $SQL$:=$SQL$ + ”AND ” + negativeAttr(r,p) +
            ”NOT IN (SELECT * FROM ” + $p$ + ”)”;
    $SQL$:=$SQL$ + ”EXCEPT (SELECT * FROM ” + head($r$) + ”)”;
    return $SQL$;
end.
```

Figure 5.5: Function TranslateRuleWithNegation

Translating rules with negated atoms.

Intuitively, the construction of the SQL statement for this kind of rules is carried out as follows: the positive part of the rule is handled in a way very similar to what has been shown in function TranslatePositiveRule; then, each negated atom is handled by a corresponding NOT IN part in the statement. The function is illustrated in Figure 5.5.

Here, *head_attr*, *body^{+}(r)*, *joinConditions*, *bodyConstantConditions* and *head* have been introduced previously. *body^{-}(r)* returns the relations corresponding to negated atoms in the body of r; *negativeAttr(r,p)* singles out those attributes of positive atoms in r bound to attributes of the negated atom p.

Example 5.4.2 The following program computes (using the goal $topEmployee$) the employees which have no other boss than the director.

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

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

```
INSERT INTO topEmployee (
    SELECT employee.att$_1$
    FROM employee, department
    WHERE (employee.att$_3$=department.att$_1$)
    AND (employee.att4=department.att2)
    AND (employee.att1, employee.att4)
    NOT IN (SELECT otherBoss.att1, otherBoss.att2 FROM otherBoss )
EXCEPT
    (SELECT * FROM topEmployee))
```

Translating rules with built-in predicates.

As pointed out in Section 2.1, in addition to user-defined predicates some comparative and arithmetic predicates are provided by the reasoning language. When running a program containing built-in predicates, the range of admissible integer values must be fixed. We map this necessity in the working database by adding a restriction based on the maximum value allowed for integer variables. Moreover, in order to allow mathematical operations among attributes, DLV^{DB} requires the types of attributes to be properly defined in the database.

The function for translating rules containing built-in predicates is a trivial variation of the function for translating positive rules and, consequently, it will not be shown. As a matter of facts, the presence of a built-in predicate in the rule implies just to add a corresponding condition in the WHERE part of the statement.

Example 5.4.3 The program:

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

is translated in the SQL statement:

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

If the variables specified in the built-in are not bound to any other variable of the atoms in the body, a #maxint value is exploited to bound that variable to its admissible range of values.

Translating rules with aggregate atoms.

In Section 2.1 we introduced the syntax and the semantics of DLP with aggregates. We have also shown that specific safety conditions must hold for each rule containing aggregate atoms, in order to guarantee the computability of the corresponding rule. As an example, aggregate atoms can not contain predicates mutually recursive with the head of the rule they are placed in; from our point of view, this implies that the truth values of each aggregate function can be computed once and for all before evaluating the corresponding rule (which can be, in its turn, recursive).

Actually, the optimization process that rewrites input programs before their execution, automatically rewrites each rule containing some aggregate atom in such a way that it follows a standard format (see also Section 5.1). Specifically, given a generic rule of the form:

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

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

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

Figure 5.6: Function TranslateAggregateRule

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

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

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

Intuitively, the objective of our translation is to create an SQL view $auxAtom\_supp$ from $auxAtom$ which contains all the attributes necessary to bind $auxAtom$ with the other atoms of the original rule and a column storing the results of the computation of f over $auxAtom$; the original aggregate atom is then replaced by this view and guard conditions are suitably translated by logic conditions between variables. At this point, the resulting rule is a standard rule not containing aggregate functions and can be then translated by one of the functions we have presented previously. The function is shown in Figure 5.6.

Here function *aggr_atom(r)* returns the aggregate atoms present in r; *aux_atom(a)* returns the auxiliary atom corresponding to $Conj$ of a and automatically generated by the optimizer. Function *bound_attr(a)* yields in output the attributes of the atom a bound with attributes of the other atoms in the rule, whereas *aggr_attr(a)* returns the attribute which the aggregation must be carried out onto (the first variable in $Vars$). *aggr_func(a)* returns the SQL aggregation statement corresponding to the aggregate function of a. Function *removeFromBody(r,a)* (resp., *addToBody(r,a)*) removes (resp., adds) the atom a from (resp., to) the rule r. Finally, *aux_atom_supp(a)* yields in output the name of the atom corresponding to the just created auxiliary view, whereas *guards(a)* converts the guard of a in a logic statement between attributes in the rule.

Example 5.4.4 Consider the following rule computing the departments which spend for the salaries of their employees, an amount greater than a certain threshold, say 100000:

$costlyDep(Dep)$:- $department(Dep, \_)$,
$\#sum\{Salary, Ename : employee(Ename, Salary, Dep, \_)\} >$ 100000.[6]

The optimizer automatically rewrites this rule as:

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

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

```
INSERT INTO aux_emp (
    SELECT employee.att$_2$, employee.att$_1$, department.att$_1$
    FROM department, employee
    WHERE department.att$_1$ = employee.att$_3$
    EXCEPT
            (SELECT * FROM aux_emp))
```

The second rule is translated in:

```
CREATE VIEW $aux\_emp\_supp$ AS (
    SELECT aux_emp.$att_3$, SUM (aux_emp.$att_1$)
    FROM aux_emp
    GROUP BY aux_emp.$att_3$)

INSERT INTO $costlyDep$ (
    SELECT department.$att_1$
    FROM department, aux_emp_supp
    WHERE department.$att_1$ = aux_emp_supp.$att_1$
            AND aux_emp_supp.$att_2$ > 100000
    EXCEPT
            (SELECT * FROM $costlyDep$))
```

Translating rules with functions

Functions introduced in the program by *external atoms* are expected to be defined as scalar stored functions in the database coupled with DLV^{DB}.

Stored functions in databases can return only one scalar value; as a consequence, DLV^{DB} adopts the convention that the last variable of the external atom corresponds to the result returned by the function call, whereas all the other variables are the input for the stored function.

Then, given an external atom $\#f(X_1, \ldots, X_n, O)$ used in a rule r, only the last variable O can be considered as an output parameter, while all the other variables must be intended as input for f. This corresponds to the function call $f(X_1, \ldots, X_n) = O$ on the DLV^{DB} working database. Moreover, O can be: *(i)* bound to other variables in r's body, *(ii)* bound to a constant,

[6]Note that Ename allows to sum also the salaries of employees earning the same amount (see the discussion on sets/multisets in [31].

(iii) a variable of r's head. Then, in the SQL statement corresponding to r, a function call is introduced in the WHERE part to implement cases *(i)* and *(ii)* and in the SELECT part to implement case *(iii)*.

As an example, consider the rule:

$$mergedNames(ID, N) \text{ :- } person(ID, FN, LN), \#concat(FN, LN, N).$$

This rule belongs to case *(iii)* above and is translated into:

```
INSERT INTO mergedNames
    (SELECT person.ID, concat(person.FN,person.LN) FROM person);
```

Note that since the grounding phase instantiates all the variables, there is no need to invoke again the functions associated with external atoms after the grounding (this is true even for disjunctive or non stratified programs). As a consequence, the handling of external atoms can be carried out completely during the grounding and, hence, within the SQL statements generated from the datalog rules.

Evaluation of programs with list terms

In our approach, list terms are handled by suitable function calls; in particular, programs containing list terms are automatically rewritten to contain only terms and function calls. Three basic operations can be singled out to handle lists: *(i)* initialization, *(ii)* packing of a term as head of a list, *(iii)* unpacking of a list in the head term and its tail.

Lists are internally handled as strings, starting (resp., ending) with a '[' (resp., ']') where terms are separated by a ','. Initialization is then implicitly implemented by the transformation of the list in a string (recall that we currently limit lists of the form $[t_1, \ldots, t_n]$ to contain only – possibly nested – lists of constants).

Packing of a list is carried out by a function `#pack` which receives a term H and a list T and returns the list L=[H|T].[7] E.g. the rule p([H|T]):-dom(H),list(T) is translated into p(L):-dom(H), list(T), #pack(H,T,L).

Handling the unpacking is a bit more tricky. In fact, the corresponding function should return two values (the head and the tail) but database stored functions can output one value only and can not have side effects on existing tables. Then, unpacking of a list must be carried out through two different calls to functions `#head` and `#tail` introduced in Section 2.1.

As an example, a rule of the form q(H):- dom(H), list(T), list([H|T]) is translated into q(H):- dom(H), list(T), list(L), #head(L,H), #tail(L,T).

The corresponding SQL statement will then be

```
INSERT INTO q (SELECT dom.H,
    FROM dom, list l1, list l2
    WHERE head(l1.L)=dom.H AND tail(l1.L)=l2.L).
```

Note that availability of #head and #tail functions allows also the manipulation of nested lists.

As a final remark, in order to simplify the evaluation process, we currently associate each occurrence of a list term in the head (resp., body) of a rule with a call to #pack (resp., #head and #tail). This may be not always the best choice in terms of efficiency, but provides a very easy way to compose multiple lists in the same rule.

[7] Here and in the following functions handling lists are supposed to be already loaded on the working database.

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

Figure 5.7: Function TranslateRecursiveRule

Translating recursive rules.

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

Intuitively, the translation algorithm must first select the proper combinations of standard and differential relations from the rule r under consideration; then, for each of these combinations, it must rewrite r in a corresponding rule r'. Each r' thus obtained is non recursive and, consequently, it can be handled by Function TranslateNonRecursiveRule. Algorithm TranslateRecursiveRule is shown in Figure 5.7.

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

It is worth noticing that the execution of the queries resulting from function TranslateRecursiveRule allows the implementation of function $EVAL\_DIFF$, which is the core of our refinement to the Semi-Naive algorithm (see the algorithm of Figure 5.2).

Example 5.4.5 Consider the situation in which we need to know whether the employee e_1 is the boss of the employee e_n either directly or by means of a number of employees $e_2, .., e_n$ such that

e_1 is the boss of e_2, e_2 is the boss of e_3, etc. Then, we have to evaluate the program:

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

containing a recursive rule. This program cannot be evaluated in one single iteration of the Semi-Naive computation. In fact, the SQL statement corresponding to the recursive rule must be executed until no new values can be derived for q_2. The SQL statement obtained by Function TranslateRecursiveRule for the second rule of this example is:

```
INSERT INTO $\Delta q_2^k$ (
    SELECT $q_2^{k-2}.att_1$, $\Delta q_2^{k-1}.att2$
    FROM $q_2^{k-2}$,$\Delta q_2^{k-1}$
    WHERE ($q_2^{k-1}.att_2$=$\Delta q_2^{k-1}.att_1$)
    EXCEPT (SELECT * FROM $\Delta q_2^k$)
    UNION
    SELECT $\Delta q_2^{k-1}.att_1$, $q_2^{k-2}.att2$
    FROM $\Delta q_2^{k-1}$, $q_2^{k-2}$
    WHERE ($\Delta q_2^{k-1}.att_2$=$q_2^{k-1}.att_1$)
    EXCEPT (SELECT * FROM $\Delta q_2^k$)
    UNION
    SELECT $\Delta q_2^{k-1}.att_1$, $\Delta q_2^{k-1}$_1$.att2$
    FROM $\Delta q_2^{k-1}$, $\Delta q_2^{k-1}$ AS $\Delta q_2^{k-1}$_1
    WHERE ($\Delta q_2^{k-1}.att_2$=$\Delta q_2^{k-1}$_1$.att_1$)
    EXCEPT (SELECT * FROM $\Delta q_2^k$))
```

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

```
EXCEPT (SELECT * FROM $\Delta q^{k-1}$)
EXCEPT (SELECT * FROM $q^{k-2}$)
```

they allow to evaluate instruction (5) of the algorithm in Figure5.2 along with function $EVAL\_DIFF$ in one single SQL statement.

Chapter 6

Applications

Coupling the expressive power of logic-based systems with the efficient data management features of DBMSs is useful in many application areas, mainly in those contexts where powerful reasoning capabilities as well as the handling of huge amounts of data are required.

In this chapter three different data-intensive scenarios are presented, including: *(i)* application to deductive databases; *(ii)* application to data-integration; *(iii)* application to RDF(S) ontology querying. These contexts compose also the test beds of our experiments presented in Chapter7.

Part of the material presented in this chapther appeared in [142, 143, 144]

6.1 Application to deductive databases

One of the most fundamental uses of a Database is to store and retrieve information, particularly when there is a large amount of data to be stored. Mining information and knowledge from large databases has been recognized by many researchers as a key research topic in database systems and machine learning fields, and by many industrial companies as an important area with an opportunity of major revenues.

A mounting wave of data intensive and knowledge based applications, such as Data Mining, Data Warehousing and Online Analytical Processing (OLAP) have created a strong demand for more powerful database languages and systems. This led to the definition of both several data model extensions (e.g., the Object Relational model), and new language constructs (e.g., recursion and OLAP constructs), and various database extenders (based, e.g., on user defined functions), to enhance the current Database Management Systems (DBMSs). A great effort in this direction has been carried out with the introduction of a new standard for SQL, namely SQL99 [126] which provides, among other features, support to object oriented databases and recursive queries. However, the adoption of SQL99 is still far from being a "standard"; in fact almost all current DBMSs do not fully support it and, in some cases, they adopt proprietary (non standard) language constructs and functions to implement parts of it. Moreover, the efficiency of current implementations of SQL99 constructs and their expressiveness are still not sufficient for performing complex reasoning tasks on huge amounts of data. On the other hand, the explosive growth of new database applications has, in several cases, outpaced the progress made by database technology.

A partial solution to these problems would be a mechanism for reasoning about the stored information. This desirable mechanism should be capable of managing and handling very large amounts of information, as well as of performing sophisticated inference tasks, and of drawing the appropriate conclusions.

The needed reasoning capabilities can be provided by logic-based systems. In fact, declarative logic programming provides a powerful formalism capable of easily modelling and solving complex problems. While research in this area initially had mainly a theoretical impact, the recent development of efficient logic-based systems like DLV [80], Smodels [101], XSB [114], ASSAT [84, 86], Cmodels [62, 61], CLASP [56], etc., has renewed the interest in the area of non-monotonic reasoning and declarative logic programming for solving real world problems in a number of application areas. As a consequence, they can provide the powerful reasoning capabilities needed to solve novel complex database problems. However, as previously pointed out, many of the interesting problems are "data intensive" and can not be handled in a typical logic programming system working in main-memory.

These considerations put into evidence that efficient and effective data management techniques combining Logic Inference Systems with Database Management Systems, are mandatory. In particular, there is the need of combining the expressive power of logic-based systems with the efficient data management features of DBMSs. Indeed, logic-based systems provide an expressive power that goes far beyond that of SQL99, whereas good DBMSs provide very efficient query optimization mechanisms allowing to handle massive amounts of data.

In the literature Deductive Database Systems (DDS) have been proposed to combine these two realities [25, 52, 23, 63]; basically, they are an attempt to adapt typical Datalog systems, which have a "smalldata" view of the world, to a "largedata" view of the world via intelligent interactions with some DBMSs. In more detail, DDSs are advanced forms of database management systems, whose query languages, based on logics, are very expressive. DDSs not only store explicit information in the manner of a relational database, but they also store rules that enable deductive inferences based on the stored data. Using techniques developed for relational systems in conjunction with declarative logic programming, deductive databases are capable of performing reasoning based on that information.

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

However, recently emerging application contexts such as the ones raising from the natural recursion across nodes in the Internet, or from the success of intrinsically recursive languages such as XML [141], renewed the interest of the scientific community in the development of efficient and flexible deductive databases systems [1, 91].

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

DLV^{DB} provides a contribution in this setting, bridging the gap between logic-based DDSs and DBMSs. It presents features of an efficient DDS but also extends the capability of handling data residing in external databases to a disjunctive logic programming system. The experimental results, presented in Section 7.1, show that DLV^{DB} significantly outperforms both commercial DBMSs and other logic-based systems in the evaluation of recursive queries.

6.2 Application to data integration

Data integration systems provide a transparent access to different and possibly distributed sources. The user is provided with a uniform view of available information by the so-called *global schema*, which queries can be posed upon. The integration system is then in charge of accessing the single sources separately and merging data relevant for the query, guided by mapping rules that specify relationships holding between the sources and the global schema [8, 78].

The global schema may contain integrity constraints (such as key dependencies, inclusion dependencies, etc.). The main issues in data integration arise when original sources independently satisfy the integrity constraints but, when they are merged through the mappings, they become inconsistent. As an example, think to the lists of students of two universities; each student has an unique ID in his university, but two different students in different universities may have assigned the same ID. Clearly, when they are loaded in a global database merging students lists, it is likely that the key constraint on student IDs of the global schema will be violated.

Most of the solutions to these problems are based on database repair approaches. Basically, a repair is a new database satisfying constraints of the global schema with minimal differences from the source data. Note that multiple repairs can be singled out for the same database. Then, answering queries over globally inconsistent sources consists in computing those answers that are true in every possible repair; these are called *consistent* answers in the literature.

DLP under ASP is a powerful tool in this context, as demonstrated for example by the approaches formalized in [8, 21, 40]. In fact, if mappings and constraints on the global schema are expressed as DLP programs, and the query Q as a union of conjunctions on the global schema, the database repairs correspond to the stable models of the program, and the consistent answers to Q correspond to the answers of Q under cautious reasoning.

As an example, the approach proposed in [40] consists in first retrieving as much information as possible from the sources, and then building the repairs by removing the minimal amount of inconsistent data and adding the minimal amount of missing data.

Example 6.2.1 To have an intuition on the repair approach proposed in [40] for handling key constraints, consider two sources s1(SID, StudentName) and s2(SID, StudentName), storing the students of two universities, and assume that the global schema is designed so as to merge these lists. The program defining the mappings for the global relation studentG and handling the key constraint over SID is:

studentD(SID,SName):- s1(SID,SName).
studentD(SID,SName):- s2(SID,SName).
studentG(SID,SName):-studentD(SID,SName), not $\overline{student}$(SID,SName).
$\overline{student}$(SID,SName1) v $\overline{student}$(SID,SName2):- studentD(SID,SName1),
studentD(SID,SName2), SName1$\neq$SName2.

Here the first two rules load all possible data from the sources, whereas the third one avoids to put conflicting tuples in the global relation studentG. Note that the disjunctive rule allows the generation of the various repairs by singling out the conflicting tuples.

Now, assume that s1 contains {s1(1234, Jhon), s1(2345, Andrew)} and s2 contains {s2(1234, David)}. There is globally a conflict between Jhon and David because they have the same ID. Then, there are two repairs for studentG, namely {studentG(1234,Jhon), studentG(2345, Andrew)} and {studentG(1234,David), studentG(2345, Andrew)}. If the user poses the query Q1(SName):-studentG(SID,SName), the only consistent answer is: {Andrew}, but if the user asks for Q2(SID):- studentG(SID,SName), the consistent answers are: {1234,2345}.

The main goal of this thesis is precisely to improve efficiency and usability of DLP systems in data-intensive contexts, like data integration, where knowledge representation and reasoning capabilities are required. In fact, the DLV^{DB} system presents *(i)* an evaluation strategy devoted to carry out as much as possible of the reasoning tasks in mass memory without degrading performances, thus allowing to deal with data-intensive applications; *(ii)* extends the expressiveness of DLP with external function calls, yet improving efficiency (at least for procedural sub-tasks) and knowledge-modelling power; *(iii)* extends the expressiveness of DLP for supporting also the management of recursive data structures (lists).

Test results, reported in Section 7.2, show that DLV^{DB} is well suited for data integration applications both for time and space requirements.

6.3 Application to RDF(S) ontology querying

The *Semantic Web* [17, 46] is an extension of the current Web by standards and technologies that helps machines understand the information on the Web. In this context, machines should be enabled to support richer discovery, data integration, navigation, and automation of tasks. Roughly speaking, the main ideas underlying the Semantic Web are oriented to *(i)* add a machine-readable meaning to Web pages and Web resources (annotations), *(ii)* use ontologies for a precise definition of shared terms in Web resources, *(iii)* make use of Knowledge Representation and Reasoning technology for automated reasoning on Web resources, and *(iv)* apply cooperative agent technology for processing the information on the Web. The development of the Semantic Web proceeds in layers of Web technologies and standards, where every layer is built on top of lower layers.

Research work is continuously ongoing on the three consecutive RDF(S) (Resource Description Framework (Schema)), Ontology and Rule layers (listed from bottom to top). The RDF(S) layer was initially conceived as a basic framework for defining resources available on the Web and their connections. RDF (Rich Description Framework), refers to a logical format of information, which is based on an encoding of data as a labeled graph (or equivalently, a ternary relation, commonly called RDF *graph*). RDF data can be interpreted under plain RDF semantics or under RDFS semantics. In the aforementioned layered vision, RDF(S) should have little or no semantics, focusing only on the logical format of information.

The Ontology layer should be built on top of RDF(S) and should provide the necessary infrastructure for describing knowledge about resources. An ontology can be written using one of the three official variants of the Web Ontology Language (hereafter OWL) [104], currently accepted as a W3C Standard Recommendation. An OWL knowledge base is written in RDF format, where some of the keywords of the language are enriched with additional meaning.

Having machine readable annotations coupled with Web resources enables a variety of applications. As a traditional example, consider flight, train, coach and metro carriers, and hotels offering their services on the Web. They might, for instance, export timetables and room availabilities in standard RDF/OWL format. An automated Web service could thus easily compose data coming from different carriers and hotels, and after performing necessary reasoning tasks (for instance, aligning arrival/departure times and room availabilities, combining "train trips" with "flight trips" once it is inferred they are specialized subclasses of "trips"), it can propose overall itinerary solutions to end users. While services offering partial solutions to this task (and similar ones) exist, it is currently hard to achieve this goal in its entirety, due to the lack of structure in Web information.

OWL is based on description logics [11]. Description logics has a long tradition as a family of formalisms for describing concept terminologies and can feature rich expressiveness, like some prominent description logics, such as $\mathcal{SHOIN}(\mathbf{D})$ (which is the theoretical basis of the variant OWL-DL of OWL). The payload of this expressiveness is, unfortunately, the high computational cost associated with many of the reasoning tasks commonly performed over an ontology. Nonetheless, a variety of Web applications require highly scalable processing of data, more than expressiveness. This puts the focus back to the lower RDF(S) data layer. In this context, RDF(S) should play the role of a lightweight ontology language. In fact, RDF(S) has few and simple descriptive capabilities (mainly, the possibility of describing and reasoning over monotonic taxonomies of objects and properties). One can thus expect from RDF(S) query systems the ability of querying very large datasets with excellent performance, yet allowing limited reasoning capabilities on the same data. In fact, as soon as the RDF(S) format for data was settled, and much earlier than when RDF(S) ontologies became growingly available[1], research has focused on how RDF(S) can be fruitfully stored, exchanged and queried[2].

As a candidate W3C recommendation [34], the SPARQL language is reaching consensus as query language of election for RDF(S) data. In this scenario, an RDF(S) storage facility (commonly called *triplestore*) plays the role of a database. However, an important difference with respect to traditional relational databases, is that a triplestore (also) represents information not explicitly stored, and which can be obtained by logical inference. Allowed logical inference is specified in terms of *entailment rules*. Different kinds of entailment rules can be exploited, namely *normative* (i.e. coming from the RDF(S) semantics specifications [139]), some subset of the normative ones (such as the so-called ρDF fragment of RDF(S) introduced in [102]) or user defined entailment rules.

Figure 6.1(a) shows the generic architecture of most current triplestore querying systems. In particular, the triplestore acts as a database and the query engine (possibly a SPARQL-enabled one[3]) manipulates this data. Most of the current query engines (with the notable exception of ARQ [9]) adopt a *pre-materialization* approach, where entailment rules are pre-computed and the initial triplestore is enriched with their consequences before carrying out any querying activity.

Unfortunately, triplestores based on the pre-materialization approach outlined above have some drawbacks:

- Inferred information is available only after the often long lasting pre-materialization step. Pre-materialization is unpractical if massive amounts of data are involved in the inferencing process; in fact, inferred information is usually much bigger in size than the original one.

[1] Several datasets are nowadays available in RDF format, such as DBLP [28] and Wikipedia [10]. Also, it is possible to convert legacy data into RDF by means of ad-hoc services [19].

[2] The reader may find in [115] a good starting point to the vast applicative and academic research currently under development for RDF.

[3] E.g. [9, 2] and [122]. The reader may refer to [125] for a thorough survey.

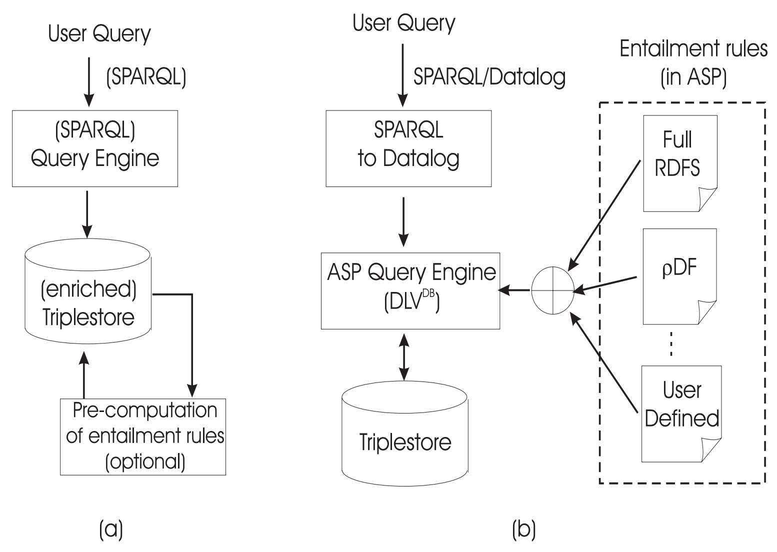


Figure 6.1: *(a)* State-of-the-art architectures for querying RDF(S) triplestores. *(b)* Our proposal.

As an example, if only the ρDF fragment of RDFS is considered, this growth has been empirically estimated (in, e.g., [127]) as more than twice the original size of the dataset. Actually, a materialized dataset can be cubically larger in theory [107]. If the full normative RDF(S) semantics is considered, then the set of inferred triples is infinite and cannot be materialized at all.

- Entailment rules are “statically” programmed by coupling a parametric reasoner (designed “ad-hoc”) with the original triplestore code. This prevents the possibility to dynamically prototype new inference rules, and to activate/de-activate inference rules depending on the given application. For instance, one might want to restrict RDF(S) inference only to the known ρDF fragment, or to enlarge inference with other (normative or not) entailment rules such as the D-entailment rules introduced in [139].

- The basic reasoning machinery of RDF(S) prescribes a heavy usage of transitive closure (recursive) constructs. Roughly speaking, given a class taxonomy, an individual belonging to a leaf class must be inferred to be member of all the ancestor classes, up to the root class. This prevents a straightforward implementation of RDF(S) over RDBMSs, since RDBMSs usually feature very primitive, and inefficient, implementations of recursion in their native query languages.

In [142, 143, 144] we presented a new architecture for querying triplestores, based on Answer Set Programming (ASP), which overcomes all the drawbacks outlined above and improves both efficiency and expressiveness of current state-of-the-art systems. The proposed architecture is shown in Figure 6.1(b). The user is allowed to express queries in both SPARQL and Datalog. A SPARQL query is first translated into Datalog by a corresponding module, which implements the approach proposed in [106], whereas a Datalog query is passed unchanged to the next module. Entailment rules are expected to be expressed in ASP, and different sets of entailment rules can be “attached” to the triplestore. As an example, in Figure 6.1(b) three sets are shown, namely full normative RDFS, ρDF and user-defined entailment rules. The user can choose on the fly, at query time, which set of entailment rules must be applied on the triplestore, and/or design his own. Note that, rather than materializing the whole output of the application of entailment rules, the rules are used as an inference mechanism for properly answering the specific input query at hand. The query engine is then implemented by means of a database oriented version of an ASP evaluator, which can both access and modify data in the triplestore.

DLV^{DB} turned out to be particularly effective for reasoning about massive ontologies and supports a rich query and reasoning language including stratified recursion, true negation, negation as failure, and all built-in and aggregate functions already introduced in DLV [45]. The experiments, proposed in Section 7.3, prove that our solution, based on DLV^{DB}, improves both scalability and expressiveness of several state-of-the-art ontology querying systems. As a consequence, DLV^{DB} is a good candidate also as an ontology querying engine.

Chapter 7

Benchmarks

In this chapther we present our experimental framework and the results; in order to asses the performance of the DLV^{DB} system, we carried out several tests from *(i)* the context of deductive databases; *(ii)* four categories of real world applications: data-integration, combinatorial problems, data transformation, and string similarity computation; and *(iii)* querying on RDF(S) ontologies context.

The chapter is organized as follows: in Sections 7.1 we present results obtained for testing in the context of deductive databases [14, 64] on several data structures, whereas, in Section 7.2 we present the results obtained in querying inconsistent and incomplete data. In Section 7.3 we present the results of our experiments aiming at comparing the performance of DLV^{DB} with several state-of-the-art ontology querying systems, whereas, in Section 7.4 we present results of testing on a combinatorial problem. Finally, in Section 7.5 and 7.6 we present results of testing functions capability.

Part of the material presented in this chapther appeared in [147, 151, 142, 143, 148, 149, 144, 152, 150]

7.1 Testing on normal stratified programs with recursion

In this section we present results obtained comparing the performance of DLV^{DB} system with several state-of-the-art systems on simple problems. Benchmarks have been designed following the guidelines, problems and data structures proposed in [14] and [64] to assess the performance of deductive database systems. Roughly speaking, problems used in [14] and [64] basically resort to the execution of some of recursive queries on a variety of data structures. The main goal of our experiments was to evaluate the deductive capabilities of tested systems for both query answering time and amount of manageable data.

7.1.1 Overview of Compared Systems

In order to provide a comparative and comprehensive analysis with state-of-the-art deductive systems, we compared DLV^{DB} with:

- SQLServer, DB2 and Oracle, because they are among the most efficient database system engines and they support the execution of recursive queries;

- LDL++, because it is one of the most robust implementations of deductive database systems;
- XSB, as an efficient implementation of the Top-Down evaluation strategy;
- Smodels, one of the most widely used Answer Set Programming systems together with DLV.

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

In the following we briefly overview the main characteristics of tested systems, focusing on their support to the language and technological capabilities addressed in this work. Specifically, we consider, for each database system, its capability to express recursive queries and, for each logic-based system, the expressiveness of its language and its capability to interact with external DBMSs. As far as database systems are concerned, it is worth pointing out that none of the considered ones fully adopt the SQL99 standard for the definition of recursive queries, but proprietary constructs are introduced by each of them.

SQL Server Microsoft SQL Server is a relational database management system produced by Microsoft. It supports the Microsoft's version of Structured Query Language (SQL). The code base for Microsoft SQL Server originated in Sybase SQL Server, and was Microsoft's entry to the enterprise-level database market, competing against Oracle, IBM, and Sybase. The current version of Microsoft SQL Server, the one we used in our tests, is denoted by *SQLServer 2005*; it uses a variant of SQL called T-SQL, or Transact-SQL, an implementation of SQL99 with support to stored procedures.

SQL Server supports the standard recursive functionalities of SQL99 that are needed for our benchmarks, even if proprietary constructs must be added in some cases to guarantee query termination.

DB2 Since the 1970s, when IBM Research invented the Relational Model and the Structured Query Language (SQL), IBM has developed a complete family of RDBMS software. Development started on mainframe platforms such as Virtual Machine (VM), Virtual Storage Extended (VSE), and Multiple Virtual Storage (MVS).

Today, DB2 represents a portfolio of information management products. The latest release of DB2 database (DB2 UDB Version 8.2), which is the one we used in our tests, is consistent with the SQL99 Core standard. In particular, DB2 supports the standard recursive functionalities that are needed for our benchmarks, even if proprietary constructs must be added to the standard SQL99 statement to guarantee the termination of some kinds of queries.

Oracle The Oracle system extends the core database functionalities with several features such as complementary software development, decision support tools, portability across platforms, and connectivity over standard networks. The latest available release of Oracle database (Oracle 10 family) provides a rich set of functions to extend the power of the database; as an example, it provides a rich toolset for manipulating data with SQL queries such as math operations, string operations, analytical functions, and XML processing.

Oracle implements a large subset of SQL99 features and supports recursion; however, as far as recursive queries are concerned, Oracle exploits proprietary constructs which do not follow the standard SQL99 notation, and whose expressiveness is lower than that of SQL99; as an example, it is not possible to express unbound queries within recursive statements (e.g., *all* the pairs of nodes linked by at least one path in a graph). In our tests we used Oracle 10.

LDL++ The LDL project [25] is directed towards two significant goals. The first one is the design of the 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-based 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.

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

XSB The XSB system [114] is an inmemory deductive database engine based on a Prolog/SLD resolution strategy called SLG. It can compute the well-founded semantics for normal logic programs. The inference engine, which is called SLG-WAM, consists of an efficient tabling engine for definite logic programs, which is extended by mechanisms for handling cycles through negation. These mechanisms are negative loop detection, delay and simplification. They serve for detecting, breaking and resolving cycles through negation.

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

SModels The SModels system [101, 100] implements the answer set semantics for normal logic programs extended by built-in functions as well as cardinality and weight constraints for domain-restricted programs.

The SModels system takes as input logic program rules in Prolog style syntax. However, in order to support efficient implementation techniques and extensions, the programs are required to be *domain-restricted* where the idea is the following: the predicate symbols in the program are divided into two classes, *domain predicates* and *non-domain* predicates. Domain predicates are predicates that are defined 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, which are similar to #`count` and #`sum` aggregates of DLV.

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

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

7.1.2 Benchmark Problems

To asses the performance of the systems described above, we carried out several tests using classical benchmark problems from the context of deductive databases [14, 64], namely *Reachability* and *Same Generation*. The former allows the analysis of basic recursion capabilities of the various systems on several data structures, whereas the latter implements a more complex problem and, consequently, allows the capability of the considered systems to carry out more refined reasoning tasks to be tested.

For each problem, we measured the performance of the various systems in computing three kinds of queries, namely: unbound queries; queries with one bound parameter; queries with all bound parameters. Considering these three cases is important because DBMSs and Deductive Databases generally benefit of query bindings (by "pushing down" selections through relational algebra optimizations, magic set techniques, or, for XSB, top down evaluation), whereas ASP systems are generally more effective with unbound queries (since they usually compute the entire models anyway); as a consequence, it is interesting to test all these systems in both their favorable and unfavorable contexts. It is worth pointing out that some of the tested systems implement optimization strategies 'a la magic set' [12, 16, 98, 116] (e.g., DLV^{DB} and LDL++), typical of deductive databases, or other program rewriting techniques; as a consequence, the actually evaluated programs are the optimized ones automatically derived by these systems, but the cost of these rewritings has been always considered in the measure of systems' performance.

In what follows we briefly introduce the two considered problems; the interested reader can find all details about them in [14].

Reachability

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

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

Same Generation

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

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

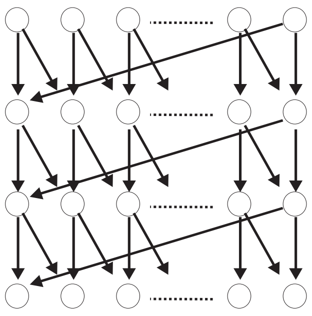


Figure 7.1: Example of a cylinder graph.

7.1.3 Benchmark Data Sets

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

Reachability As for the Reachability problem, we considered: *(i)* full binary trees, *(ii)* acyclic graphs (a-graphs in the following), *(iii)* cyclic graphs (c-graphs in the following), and *(iv)* cylinders [14].

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

Cylinders are particular kinds of acyclic graphs which can be layered; each layer has the same number of nodes and each node has two incoming and two outgoing arcs. An example of a cylinder is shown in Figure 7.1. A cylinder can be characterized by a width and a height. The ratio $\rho = \frac{\text{width}}{\text{height}}$ can be exploited to characterize a cylinder; we generated various categories of cylinders characterized by values of ρ equal to 0.5, 1.0 and 1.5 respectively. Due to space constraints, in this paper we report just the results obtained for $\rho = 1$.

Graphs have been generated using the Stanford GraphBase [73] library whereas trees and cylinders have been generated using ad-hoc procedures, since they are characterized by a regular structure.

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

7.1.4 Results and Discussion

The encodings of the problems we have exploited in our tests are provided in the AppendixA. Note that, since Oracle does not support the standard SQL99 language, but only a simplified form of recursion, we have not tested this system along with the other ones. We will discuss encodings and results obtained for Oracle in a separate section.

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

In these tests we measured the time required by each system to answer the various queries. We fixed a maximum running time of 12000 seconds (about 3 hours) for each test. In the following figures, the line of a system stops whenever some query was not solved within this time limit (note that graphs have a logarithmic scale on the vertical axis).

In more detail, Figure 7.2 shows results obtained for the Same Generation problem, whereas Figure 7.3 (resp., 7.4, 7.5, 7.6) presents the results measured for the Reachability problem when applied on acyclic graphs (resp., cyclic graphs, cylinders, trees).

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

In particular, in some tests, XSB shows a good behaviour (e.g., in Reachability on cyclic graphs and cylinders) but, even in those positive tests, it "dies" earlier than DLV^{DB} (with the exception of reachable(b1,Y) on cylinders), probably because it exceeds the main-memory.

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

DB2 performance is near to that of DLV^{DB} only in samegen(X,Y); in all the other cases its line is near to the vertical axis.

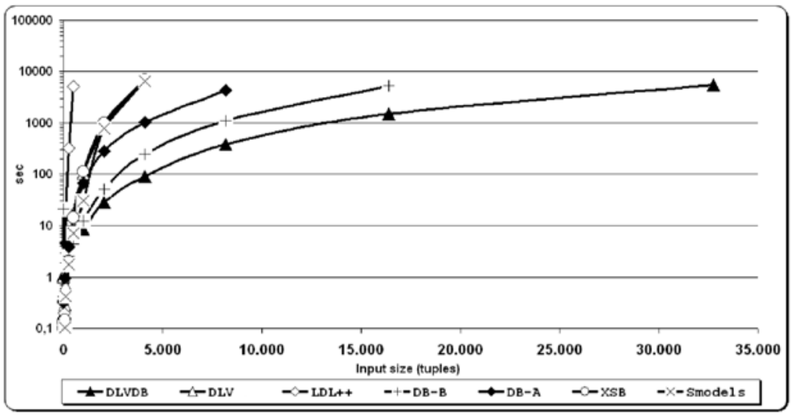
SQLServer showed very good performance only for reachability on trees (see also Table7.1 introduced next). This behaviour could be justified by the presence of optimization mechanisms implemented in this system which are particularly suited for computing the transitive closure on simple data structures (like trees), but these are not effective for other (more complex) kinds of queries/data structures.

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

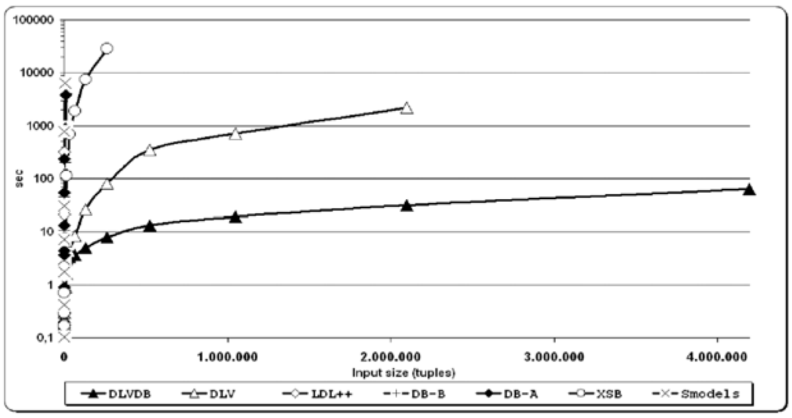
Finally, as expected, DLV allows to handle lower amounts of data w.r.t. DLV^{DB}; however, in several cases it was one of the best three performing systems, especially on bound queries. This result is mainly due to the magic sets optimization technique it implements.

A rather surprising result is that DLV has almost always higher execution times than DLV^{DB} even for not very high input data sizes. The motivation for this result can be justified by the following reasoning. Both DLV^{DB} and DLV benefit of all the program rewriting optimization techniques developed in the DLV project; moreover, both of them implement a differential Semi-Naive approach for the evaluation of normal stratified programs. However, while DLV reasons about its underlying data in a tuple-at-a-time way, DLV^{DB} exploits set-at-a-time strategies (implemented by SQL queries); this, in conjunction with the fact that the underlying working database implements advanced optimization strategies for executing joins, makes DLV^{DB} more efficient than DLV even when all the data fits in main-memory.

As pointed out also in [14], another important parameter to measure in this context is the system's capability of handling large amounts of input data. In order to carry out this kind of verification, we considered the time response of each system for the largest input data set we samegen(X,Y)



samegen(b1,Y)



samegen(b1,b2)

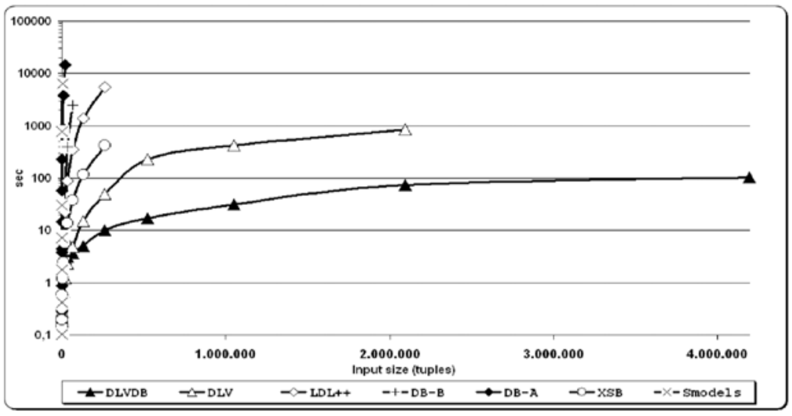
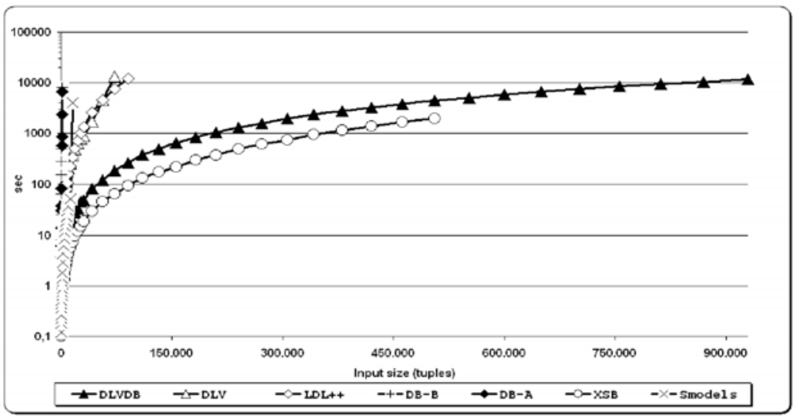
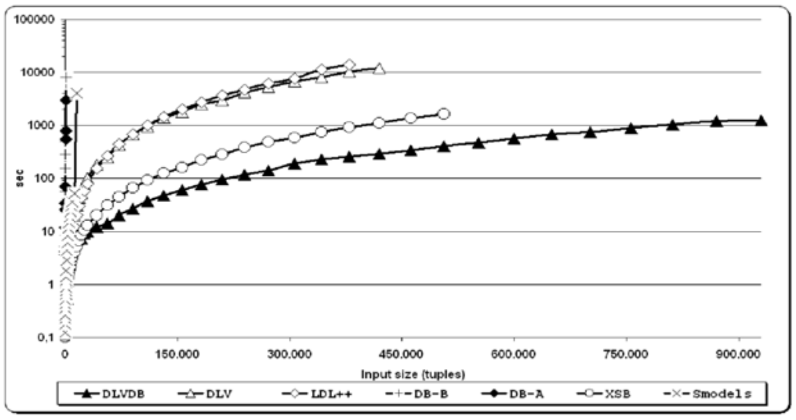


Figure 7.2: Results for Same Generation on trees

reachable(X,Y) on a-graphs



reachable(b1,Y) on a-graphs



reachable(b1,b2) on a-graphs

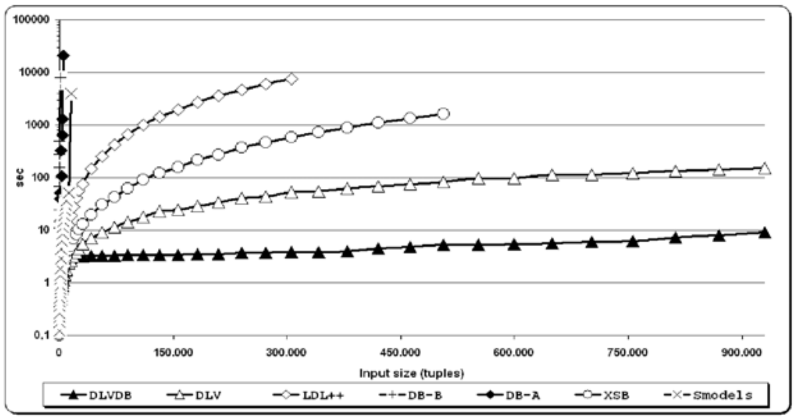
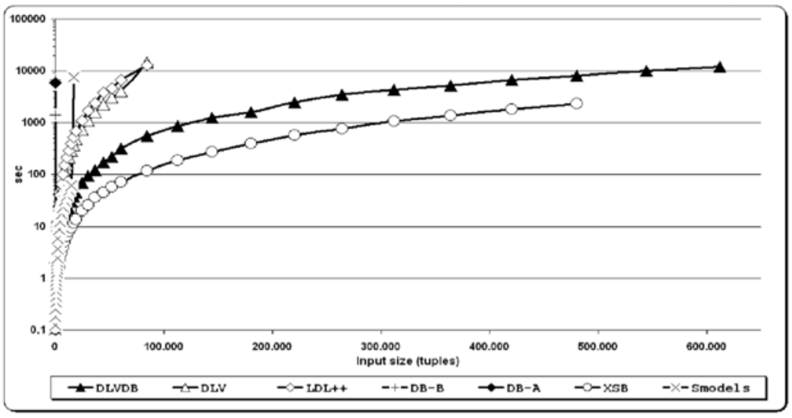
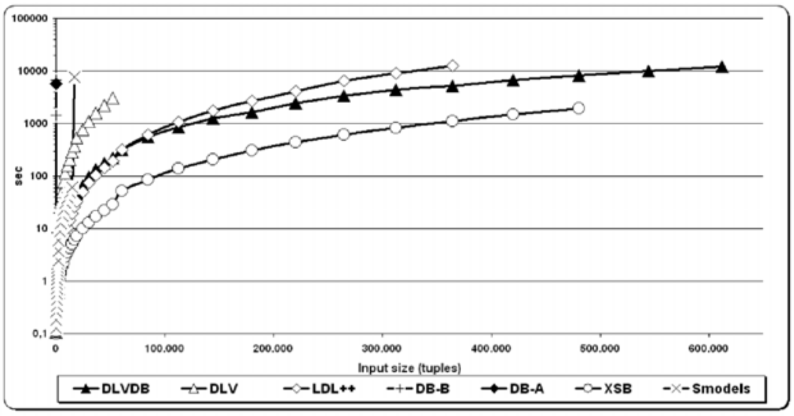


Figure 7.3: Results for Reachability with acyclic graphs

reachable(X,Y) on c-graphs



reachable(b1,Y) on c-graphs



reachable(b1,b2) on c-graphs

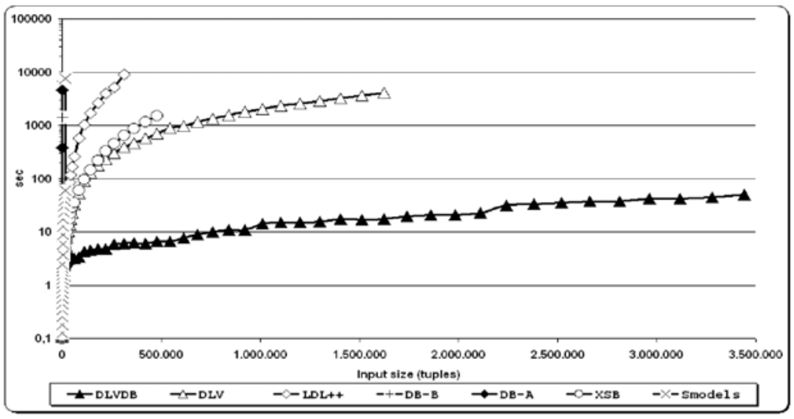
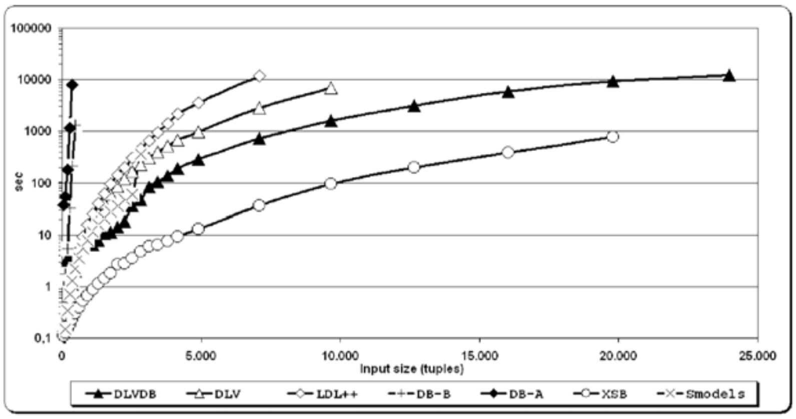
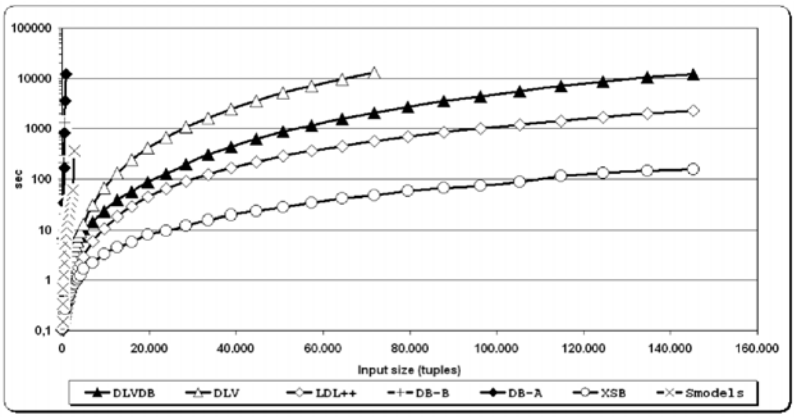


Figure 7.4: Results for Reachability with cyclic graphs

reachable(X,Y) on cylinders



reachable(b1,Y) on cylinders



reachable(b1,b2) on cylinders

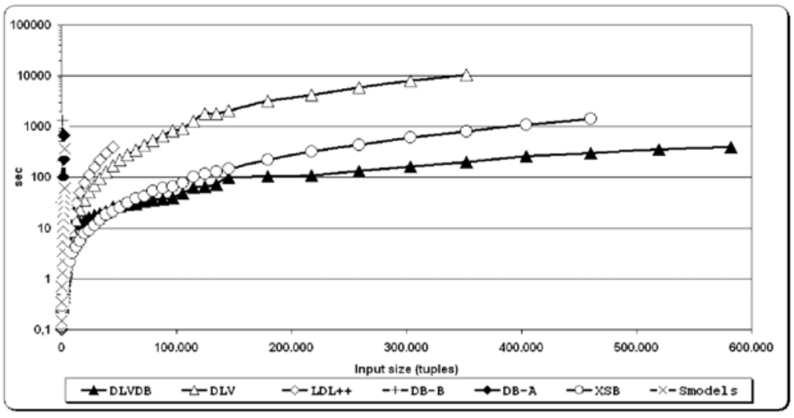
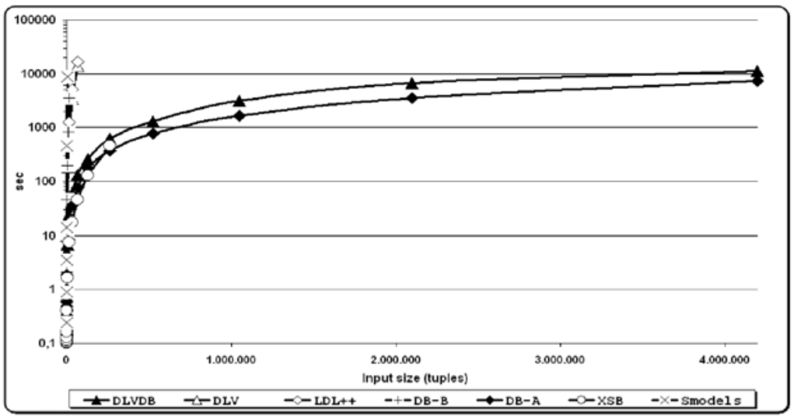
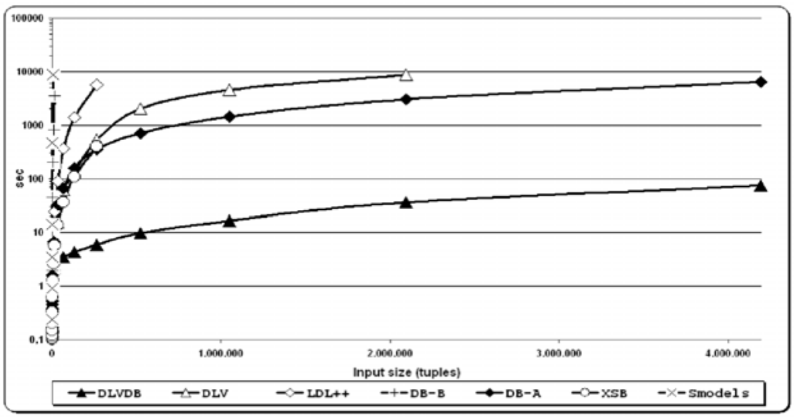


Figure 7.5: Results for Reachability with cylinders

reachable(X,Y) on trees



reachable(b1,Y) on trees



reachable(b1,b2) on trees

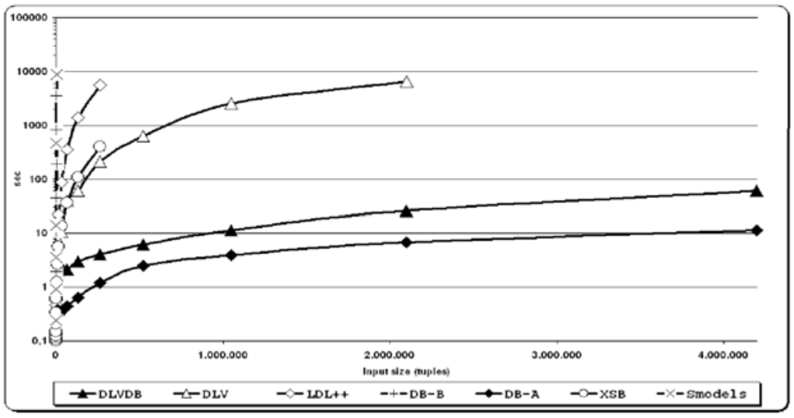


Figure 7.6: Results for Reachability with trees

have used in each query.

Table 7.1 shows the execution times measured for those systems which have been capable of solving the query within the fixed time limit of 12000 seconds; the second column of the table shows, for each query, both the input data size, measured in terms of the number of input facts (tuples), and the total amount of handled data, measured in Mbytes, given by the size of the answer set produced by DLV^{DB} in answering that query[1].

From the analysis of this table, we may observe that: *(i)* DLV^{DB} has been always capable of solving the query on the maximum data size; *(ii)* in 11 queries out of 15 DLV^{DB} (in one case along with DLV) has been the only system capable of completing the computation within the time limit; *(iii)* DLV^{DB} allowed to handle up to 6.7 Gbytes of data in samegen(X,Y) and 1.6 Gbytes in reachable(X,Y) on trees within the fixed time limit of 12000 seconds and never ended its computation due to lack of memory, as instead other systems did.

Query / *Data Type*	Input Size (tuples) / *Output size* (*Mbytes*)	DB2 (sec)	DLV (sec)	DLV^{DB} (sec)	LDL++ (sec)	Smodels (sec)	SQLServer (sec)	XSB (sec)
samegen(X,Y) *tree*	32766 *6716 Mb*	–	–	5552	–	–	–	–
samegen(b1,Y) *tree*	4194302 *78 Mb*	–	–	64	–	–	–	–
samegen(b1,b2) *tree*	4194302 *78 Mb*	–	–	102	–	–	–	–
reachable(X,Y) *a-graph*	929945 *103 Mb*	–	–	11820	–	–	–	–
reachable(b1,Y) *a-graph*	929945 *38 Mb*	–	–	1191	–	–	–	–
reachable(b1,b2) *a-graph*	929945 *17 Mb*	–	–	4	–	–	–	–
reachable(X,Y) *c-graph*	612150 *68 Mb*	–	–	11936	–	–	–	–
reachable(b1,Y) *c-graph*	612150 *68 Mb*	–	–	11933	–	–	–	–
reachable(b1,b2) *c-graph*	612150 *11 Mb*	–	981	8	–	–	–	–
reachable(X,Y) *cylinder*	23980 *465 Mb*	–	–	11784	–	–	–	–
reachable(b1,Y) *cylinder*	145260 *279 Mb*	–	–	11654	2284	–	–	157
reachable(b1,b2) *cylinder*	582120 *13 Mb*	–	–	388	–	–	–	–
reachable(X,Y) *tree*	4194302 *1634Mb*	–	–	11161	–	–	7280	–
reachable(b1,Y) *tree*	4194302 *79 Mb*	–	–	76	–	–	6438	–
reachable(b1,b2) *tree*	4194302 *78 Mb*	–	–	60	–	–	12	–

Table 7.1: Execution times of the systems capable of solving the query for the maximum considered size of the input data

Comparison to Oracle

As previously pointed out, Oracle does not support the standard SQL99 encoding for recursive queries, but it exploits a proprietary language for implementing a simplified form of recursion.

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

This language is less expressive than SQL99 for recursion; as an example, unbound recursive queries cannot be implemented in Oracle; analogously, it does not allow to write recursive views in a "uniform" way (i.e., independently from the specific bound parameters).

As for the problems addressed in this paper, it was not possible to write the unbound query neither for Reachability, nor for Same Generation with Oracle. The other queries have encodings not equivalent to the general version we adopted for the other systems.

As an example, the query *reachable(b1,Y)* can be expressed in Oracle by the following statement:

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

which, however, is equivalent to the datalog program:

$reached(b1).$

$reached(X)$:- $reached(Y),\ edge(Y,X).$

$reachable(b1,\ Y)$:- $reached(Y).$

This is clearly a program that can be evaluated more easily than the general encoding, because it involves a recursive rule with one single attribute and a unique starting point for the recursion (the fact *reached(b1)*); however, this query (and the equivalent program) is less general than the one introduced in Section A, since its structure must be modified if, for example, we need to carry out a query with both the parameters bound or if we want to bound the second parameter instead of the first.

Clearly, testing such encodings against the other, more general, ones would have been unfair. Anyway, we carried out some tests involving Oracle, by applying its encodings and the corresponding datalog programs on the maximum data instances we considered for the various queries, in order to have a rough idea on the performance. As an example, for the query *reachable(b1,Y)* mentioned above, on a-graphs (resp., c-graphs) of size 929945 (resp., 612150) tuples we have measured that Oracle takes 22.5 (resp., 15.9) seconds, whereas DLV^{DB} takes 6.4 (resp., 5.6) seconds. Analogously, for the query *samegen(b1,Y)*, on trees of size 4194302 tuples, Oracle requires 1329.4 seconds to terminate the computation, whereas DLV^{DB} 500.8 seconds. Oracle performed better than DLV^{DB} only for Reachability on trees; also in this case, as we have done for SQLServer, we may conjecture that this behaviour is motivated by the particular optimization techniques implemented in the system.

These results are representative of the overall performance we have measured for Oracle in our benchmarks; on one hand they confirm our claim that the encodings solvable by Oracle are very different, also from a performance point of view, w.r.t. the general ones used in our benchmarks (as an example, this is proved by the significantly lower timing measured for DLV^{DB} in $reachable(b1,Y)$ w.r.t. the same query in the standard encoding); on the other hand, they allow us to conclude that the same reasoning as that drawn in Section7.1.4 about DLV^{DB} performance is still valid.

7.2 Testing on a real data integration setting

In this section we present the results obtained in querying inconsistent and incomplete data. We exploited the data integration framework developed in the INFOMIX project (IST-2001-33570) [40] which integrates real data from a university context and we already presented in Section6.2.

7.2.1 Overview of compared systems

We compared DLV^{DB} with state-of-the-art ASP systems, namely DLV [80], Gnt2 [68], ClaspD [54], Smodels [101], and Cmodels [61]. DLV^{DB} and DLV include an internal proprietary grounder, whereas the other systems require an external grounder; we tested both Lparse [128] and GrinGo [57] for this purpose; precisely, given a grounder x and a system y, we run $x|y$ so as to direct the output of x into y; the output of the systems have been directed to null in order to eliminate printing times from the computation of the overall execution times.

It is worth pointing out that all systems but DLV^{DB} and DLV do not explicitly support non-ground queries; in order to carry out our tests, we asked these systems to compute all answer sets. However, since tested queries are all non-ground (see below) answer sets must be all computed anyway. Note also that Smodels and GrinGo do not support disjunction; since the data integration framework required some disjunctive rules for handling data inconsistencies, we adopted a semantic preserving rewriting when using these systems to remove disjunctions[2].

7.2.2 Tested queries

We tested four queries, ranging from simple selections to more complicated ones. Two of these queries have been also used for studying the scalability of tested systems:

- IQ_1: select the student IDs and the course descriptions of the examinations they passed (this query involves possible inconsistencies in student IDs, exam records, and course descriptions).
- IQ_2: select the first and second names of the professors stored in the database (this query involves possible inclusion dependency violations in relationships involving professors, and possible inconsistencies in exam records).
- IQ_3: select pairs of students having at least one common exam (this query involves possible inconsistencies in student IDs and exam records). We leveraged the complexity of this query by filtering out different subsets of exam records.
- IQ_4: select pairs of students and course codes of passed examinations such that the professor's first name of the corresponding courses is the same (this query involves possible inconsistencies in student IDs, exam records, and course descriptions). We leveraged the complexity of this query by filtering out different subsets of exam records.

All tested queries are non-ground. The complete encodings of tested queries are provided in the Appendix A.

[2]We used ClaspD also for non disjunctive programs with GrinGo. However, we checked that running times of Clasp are the same as those of ClaspD in these queries.

7.2.3 Results and discussion

Test results are shown in Figure 7.7. In the graphs, we used the notation x:y to denote the system y coupled with the grounder x; moreover, to simplify the notation, we used symbol L (resp. G) to denote Lparse (resp. GrinGo).

Results of queries IQ_1 and IQ_2 are shown in Figure 7.7(a). We can observe that the amount of data involved by these queries is still manageable by all tested systems in main memory. DLV^{DB} and DLV present comparable performances and they are at least 50% faster than other systems. In these queries, there is no substantial difference in using Lparse or GrinGo.

The scalability of query IQ_3 is illustrated in Figure 7.7(b). Here (and in Figure 7.7(c)) the line of a system stops when it (or the associated grounder) has not been able to solve the query. Note that no system but DLV^{DB} has been capable of handling 100% of input data, due to lack of memory. Specifically, for this query, grounders were able to complete the computation, but systems not. As for obtained results, it is possible to observe that in this query, when coupled with GrinGo, systems behave generally better than with Lparse, at least for small inputs. Performances of DLV^{DB} are comparable to those of the other systems with Lparse for small inputs, but it behaves much better for bigger data sizes. Notably ClaspD with GrinGo presents the best performance for IQ_3 until it is able to handle data in main memory.

Results for query IQ_4 are shown in Figure 7.7(c). Here, Lparse has not been able to complete the grounding in reasonable time even for the smallest data set (we stopped it after 12 hours). Hence, only results with GrinGo are presented (which has been able to complete the grounding for plotted data). Here, again, DLV^{DB} allows handling bigger data sizes than the other systems which, at some point, are subject to memory overflow. Also, the performances of DLV^{DB} in small data sets are extremely competitive.

Finally, Table 7.2 summarizes the biggest data sets handled by each system for queries IQ_3 and IQ_4 (the 0% in IQ_4 are due to Lparse fault).

			Lparse:				GrinGo:			
	DLV^{DB}	DLV	Gnt2	ClaspD	Smodels	Cmodels	Gnt2	ClaspD	Smodels	Cmodels
IQ_3	100%	90%	46%	57%	70%	57%	57%	90%	84%	57%
IQ_4	100%	93%	0%	0%	0%	0%	24%	93%	78%	78%

Table 7.2: Biggest data sets handled by tested systems for IQ_3 and IQ_4.

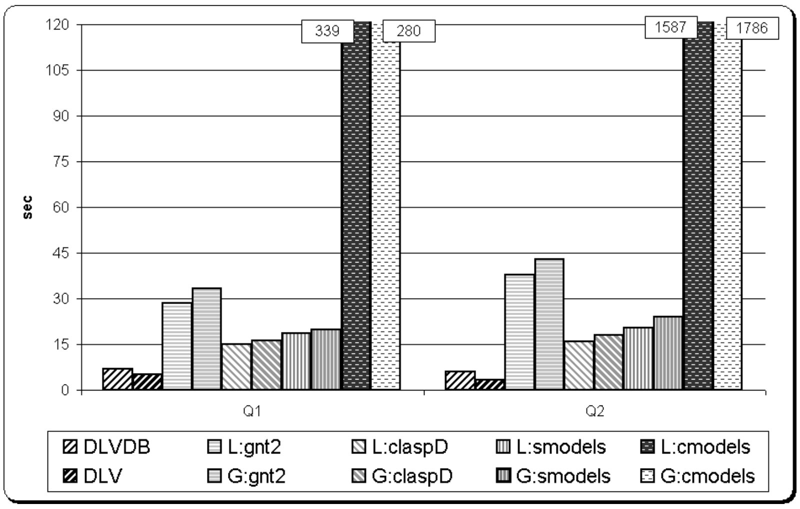
7.3 Testing on querying of RDF(S) ontologies

In this section, we present the results of our experiments aiming at comparing the performance of DLV^{DB} with several state-of-the-art triplestores in the context of ontology querying already presented in Section 6.3. The main goal of these experiments was to evaluate both the scalability and the query language expressiveness of tested systems.

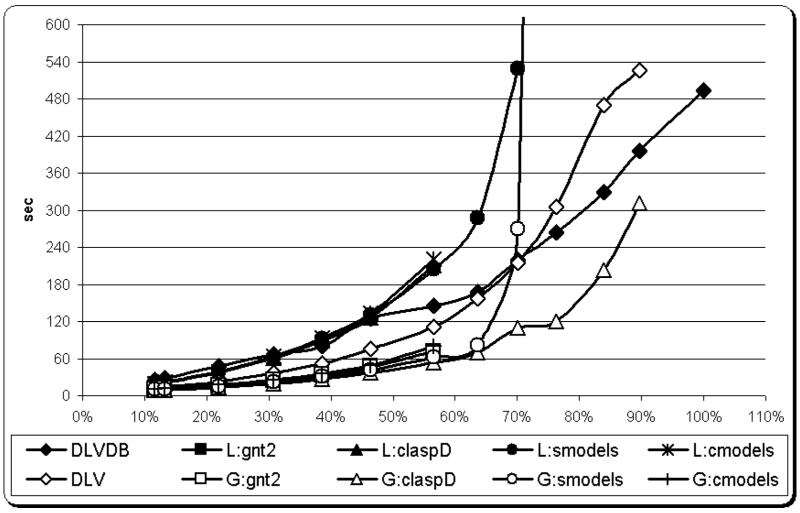
7.3.1 Compared Systems

In our tests we compared DLV^{DB} with three state-of-the-art triplestores, namely: Sesame [122], ARQ [9], and Mulgara [2]. The first two systems allow both in-memory and RDBMS storage and, consequently, we tested them on both execution modalities. In the following, we refer the

(a)



(b)



(c)

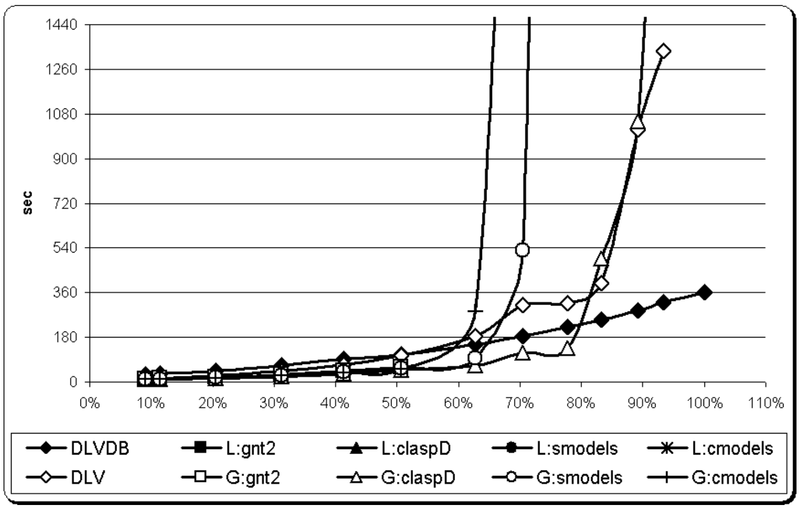


Figure 7.7: Results for : (a) IQ_1 and IQ_2; (b) IQ_3; (c) IQ_4.

in-memory version of Sesame (resp. ARQ) as Sesame-Mem (resp. ARQ-Mem) and the RDBMS version as Sesame-DB (resp. ARQ-DB). RDBMS versions of all systems (including DLV^{DB}) have been tested with Microsoft SQL Server 2005 as underlying database. As it will be clear in the following, we also tested a version of Sesame which works on files; we refer this version of Sesame as Sesame-File. For each system we used the latest available stable release at the time of writing. We next briefly describe them.

Sesame is an open source Java framework with support for storage and querying of RDF(S) data. It provides developers with a flexible access API and several query languages; however, its native language (which is the one adopted in our tests) is SeRQL – Sesame RDF Query Language. Actually, the current official release of Sesame does not support the SPARQL language yet. Some of the query language's most important features are: *(i)* expressive path expression syntax that match specific paths through an RDF graph, *(ii)* RDF Schema support, *(iii)* string matching. Furthermore, it allows simplified forms of reasoning on RDF and RDFS. In particular, inferences are performed by pre-materializing the closure $R(G)$ of the input triplestore G.

The latest official release is version 1.2.7. However, during the preparation of this manuscript, version 2.0 of Sesame became available as Release Candidate and, in order to guarantee fairness in the comparison, we considered also this version in our tests. In fact, version 2.0 supports the SPARQL Query Language and features an improved inferencing support, but does not support RDBMS management yet, allowing only files. In the following, we indicate Sesame 1.2.7 as `Sesame1` and Sesame 2.0 as `Sesame2`.

ARQ is a query engine for Jena (a framework for building Semantic Web applications, which is distributed at `http://jena.sourceforge.net`) that supports the SPARQL Query language. ARQ includes a rule-based inference engine and performs non materialized inference. As for Sesame, ARQ can be executed with data loaded both in-memory and on an RDBMS. We executed SPARQL queries from Java code on the latest available version of ARQ (2.1) using the Jena's API in both execution modalities[3].

Mulgara is a database system specifically conceived for the storage and retrieval of RDF(S) (note that it is not a standard relational DBMS). Mulgara is an Open Source active fork of the Kowari project[4]. The adopted query language is $iTQL$ (Interactive Tucana Query Language), a simple SQL-like query language for querying and updating Mulgara databases. A compatibility support with SPARQL is declared, yet not implemented. The Mulgara Store offers native RDF(S) support, multiple databases per server, and full text search functionalities. The system has been tested using its internal storage data structures (XA Triplestore). The latest release available for Mulgara is mulgara-1.1.1.

7.3.2 Benchmark Data Set

In order to provide an objective and comprehensive set of tests we adopted two different data sets: one coming from real data, that is, the DBLP database [82] and one coming from synthetic information, i.e. the LUBM benchmark suite [93].

[3]Distributed at `https://jena.svn.sourceforge.net/svnroot/jena/ARQ/`

[4]`http://www.kowari.org/`

DBLP is a real database containing a large number of bibliographic descriptions on major computer science journals and proceedings; the DBLP server indexes more than half a million articles and features several thousand links to home pages of computer scientists. Recently, an OWL ontology has been developed for DBLP data. A corresponding RDF snapshot has been downloaded at the Web address `http://sw.deri.org/∼aharth/2004/07/dblp/`. The main classes represented in this ontology are *Author*, *Citation*, *Document*, and *Publisher*, where a *Document* can be one of: Article, Book, Collection, Inproceedings, Mastersthesis, Phdthesis, Proceedings, Series, WWW. In order to test the scalability of the various systems we considered several subsets of the entire database, each containing an increasing number of statements and constructed in such a way that the greater sets strictly contain the smaller ones. Generated data sets contain from 70000 to 2 million RDF triples.

The Lehigh University Benchmark (LUBM) has been specifically developed to facilitate the evaluation of Semantic Web triplestores in a standard and systematic way. In fact, the benchmark is intended to evaluate the performance of those triplestores with respect to extensional queries over large data sets that commit to a single realistic ontology. It consists of a university domain ontology with customizable and repeatable synthetic data. The LUBM ontology schema and its data generation tool are quite complex and their description is out of the scope of this paper. We used the Univ-Bench ontology that describes (among others) universities, departments, students, professors and relationships among them. The interested reader can find all information in [93]. Data generation is carried out by the Univ-Bench data generator tool (UBA) whose main generation parameter is the number of universities to consider. Also in this case, we generated several data sets, each containing an increasing number of statements and constructed in such a way that the greater sets strictly contain the smaller ones. Generated data sets are named as: `lubm-5`, corresponding to 5 universities and about 640000 RDF triples, `lubm-10` (10 universities, about 1 million triples), `lubm-15` (15 universities, about 2 million triples), `lubm-30` (30 universities, about 4 million triples), `lubm-45` (45 universities, about 6 million triples).

7.3.3 Tested Queries

As previously pointed out, the expressiveness of the query language varies for each tested system. In order to compare both scalability and expressiveness, we exploited queries of increasing complexity, ranging from simple selections to queries requiring different forms of inferences over the data. The encodings of all the tested queries are provided in the AppendixA.

Queries on DBLP We ran the following five queries over DBLP:

- OQ_1: Select the names of the Authors and the URI of the corresponding Articles they are author of.
- OQ_2: Select the names of the Authors which published at least one Article in year 2000.
- OQ_3: Select the names of the Authors which are creators of at least one document (i.e. either an Article, or a Book, or a Collection, etc.).
- OQ_4: For each author in the database, select the corresponding name and count the number of Articles he published.

- OQ_5: Given a pair of Authors A_1 and A_2, compute the "collaborative distance" between them; the collaborative distance can be computed as the minimum length of a path connecting A_1 and A_2 in a graph where Authors are the nodes of this graph and an edge between A_i and A_j indicates that A_i co-authored at least one document with A_j.

Here, queries OQ_1 and OQ_2 are simple selections; OQ_3 requires a simple form of inference; in fact articles, books, etc. must be abstracted into documents. Query OQ_4 requires the capability to aggregate data. Finally, OQ_5 requires to perform a transitive closure over the underlying data and, consequently, the corresponding query must be recursive.

It is worth observing that queries OQ_1, OQ_2, and OQ_3 can be executed by all the evaluated systems. As for OQ_3, we exploited the Krule engine for Mulgara, the inferencing repository in Sesame and the inferencing Reasoner in ARQ. Note that Mulgara and Sesame-DB materialize the possible inferenced data just during the loading of the RDF dataset in the database; however, in our tests, we measured only the query answering times. Query OQ_4 cannot be evaluated by Sesame because the SeRQL query language does not support aggregate operators. Finally, query OQ_5 can be evaluated only by DLV^{DB}, because it is the only system allowing for recursive queries.

Queries on LUBM The LUBM benchmark provides 14 test queries. Many of them are slight variants of one another. Most of the queries basically select subsets of the input data and require, in some cases, basic inference processes (e.g. class-subclass inferences). Few of them are intended to verify the presence of certain reasoning capabilities (peculiar of OWL ontologies rather than RDFS) in the tested systems; in fact, they require the management of transitive properties. However, some important querying capabilities, such as data aggregation and recursion, are not addressed by the queries in the LUBM benchmark. As a consequence we designed also other queries over the LUBM data set to test these higher expressive features provided by DLV^{DB} and some other systems.

Queries taken from the LUBM benchmark are listed below (in parentheses, we give the official query number as given by LUBM):

- OQ_6 (LUBM Query1): Select Graduate Students attending a given Course. It is assumed that no hierarchy information or inference exists or is required (that is, no RDFS entailment is required).

- OQ_7 (LUBM Query2): Select the Graduate Students X, Universities Y and Departments Z such that X graduated in Y and is a member of Z, and Z is a sub-organization of Y. Note that this query involves three classes and three properties; moreover, there is a triangular pattern of relationships between the objects involved.

- OQ_8 (LUBM Query3): Select the Publications of a given Assistant Professor. Here Publication has a wide hierarchy of subclasses, subject to RDFS entailment.

- OQ_9 (LUBM Query4): Select the Professors working for a specific Department, showing their names, telephone numbers and email addresses. This query covers multiple properties of the single class Professor, having many subclasses, and requires a long join pattern.

- OQ_{10} (LUBM Query5): Select the Persons which are member of a specific Department; this query assumes a *subClassOf* relationship between Person and its subclasses and a *subPropertyOf* relationship between memberOf and its subproperties.

- OQ_{11} (LUBM Query6): Select all instances of the class Student, considering also its subclasses Graduate Student and Undergraduate Student.
- OQ_{12}: (LUBM Query7): Select the Students taking some Course held by a specific Associate Professor.
- OQ_{13}: (LUBM Query8): Select the Students having an email address, which are member of a Department that is a sub-organization of a given University.
- OQ_{14} (LUBM Query9): Select the Students attending some Course held by a teacher who is also their advisor. This query involves the highest number of classes and properties in the query set; moreover, there is a triangular pattern of relationships.
- OQ_{15} (LUBM Query14): Select all the Undergraduate Students. This query is very simple, yet characterized by a large input and a low selectivity.[5]

Additional queries that we have tested, not included in the LUBM benchmark, are listed below.

- OQ_{16}: For each Author in the database count the number of Papers she/he published. This query requires the capability to aggregate data.
- OQ_{17}: Given a pair of authors A_1 and A_2, compute the "collaborative distance" between them (see query OQ_5 for the definition of collaborative distance). This query requires the capability to express recursion.

7.3.4 Results and Discussion

Results on DBLP Figures 7.8 and 7.9 show the results we have obtained for the DBLP queries. In the figures, the chart of a system is absent whenever it is not able to solve the query due to some system's fault or if its response time is (except for OQ_5) greater than 3600 seconds (1 hour). Moreover, if a system's query language is not sufficiently expressive to answer a certain query, it is not included in the graph. From the analysis of the figures, we can draw the following observations.

DLV^{DB} shows always the best performance for all the queries, yet it is characterized by the highest language expressiveness. Mulgara and ARQ have, after DLV^{DB}, the more expressive query language; however, Mulgara is not able to complete the computation of OQ_3 and OQ_4 already after 220000 triples and is not able to express OQ_5, whereas ARQ always shows higher time responses than both DLV^{DB} and Mulgara, except for OQ_4 where it performs sensibly better than Mulgara. Sesame turns out to be competitive in version Sesame2-File, especially for the biggest data sets in OQ_1 and OQ_3, but scores the worst performance among all the systems for all queries in version Sesame1-DB. As far as Sesame is concerned, it is particularly interesting to observe the very different behaviour of its various versions; in particular, as for Sesame1 the in-memory version works much better than its DB version; vice versa, the file version of Sesame2 performs much better than its in-memory version. In any case, OQ_4 and OQ_5 could not be tested with Sesame due to lack of language expressiveness.

Results on LUBM Figures 7.10, 7.11, 7.12 and 7.13 show the results we have obtained for the LUBM queries. In the figures, the chart of a system is absent whenever it is not able to solve the query due to some system's fault, or if its response time is (except for OQ_{17}) greater than 3600 seconds (1 hour). The maximum allowed size for pre-materialized inference has been set to 4Gb (note that the greatest considered triple set occupied 1Gb in the database). Finally, if a system's query language is not sufficiently expressive to answer a certain query, it is not included in the graph.

System/Query	OQ_6	OQ_7	OQ_8	OQ_9	OQ_{10}	OQ_{11}
DLV^{DB}	Y	Y	Y	Y	Y	Y
Mulgara	Y	Y	Y			
Sesame1-Mem	Y	Y	Y	Y	Y	Y
Sesame1-DB	Y	Y	Y	Y		
Sesame2-Mem	Y	Y	Y	Y	Y	Y
Sesame2-File	Y	Y	Y	Y	Y	Y
ARQ-Mem	Y		Y	Y		Y
ARQ-DB						

	OQ_{12}	OQ_{13}	OQ_{14}	OQ_{15}	OQ_{16}	OQ_{17}
DLV^{DB}	Y	Y	Y	Y	Y	Y
Mulgara				Y	Y	
Sesame1-Mem	Y	Y	Y	Y		
Sesame1-DB				Y		
Sesame2-Mem	Y	Y	Y	Y		
Sesame2-File	Y	Y	Y	Y		
ARQ-Mem				Y	Y	
ARQ-DB						

Table 7.3: Summary of the systems capable of computing the results for the data sets in LUBM queries, under the specified constraints. Y=Yes, blank=No.

Table 7.3 summarizes the observed capability of each system to complete the computation of query results over all the considered data sets, under the limitations specified above.

From the analysis of Figures 7.10, 7.11, 7.12 and 7.13 and of Table 7.3, we can draw the following observations. DLV^{DB} is the only system capable of completing the computation of all the queries under the time and space constraints specified previously. Only Sesame is competitive with DLV^{DB} on this aspect; on the other hand, queries OQ_{16} and OQ_{17} cannot be computed by this system due to lack of language expressiveness.

DLV^{DB} scores also the best performance over all queries, except query OQ_{14} where Sesame has the best time responses. As far as this aspect is concerned, however, it is worth recalling that Sesame and Mulgara pre-materialize inferences during loading of data into the repository; as previously pointed out, pre-materialization times are not considered in the graphs, since this task can be carried out once for each data set[6]. For the sake of completeness, however, we show in Table 7.4 the times (in minutes) required by the various systems for loading test data in the corresponding repositories; for systems pre-materializing inferences, we distinguish between the loading of data with and without pre-materialization. From this table and Table7.3 we can also

[5]A more exhaustive description of these queries can be found at `http://swat.cse.lehigh.edu/projects/lubm/query.htm`.

[6]Recall that OQ_9-OQ_{14} are indeed the LUBM queries requiring inference.

observe that Mulgara is not able to compute inference in a reasonable time (we stopped it after 14 hours) already for `lubm-10`; this caused it to be unable to compute queries OQ_9-OQ_{14} for data sets `lubm-10`, `lubm-15`, `lubm-30`, `lubm-45`.

	DLV	ARQ	Sesame1-DB		Mulgara		Sesame2-File	
Triples				**inference**		**inference**		**inference**
`lubm-5`	5	10	13	45	5	46	3	12
`lubm-10`	10	21	28	116	14	**	5	30
`lubm-15`	15	108	61	221	30	**	9	50
`lubm-30`	31	*	102	869	106	**	19	138
`lubm-45`	46	*	161	*	196	**	31	261

Table 7.4: Loading times (minutes) for each system and data set. Whenever needed we distinguish between loading with and without inference pre-materialization. *: DB space exceeded. **: More than 14 hours.

As for the behaviour of the other systems, it is worth noting that for the LUBM data sets DB versions of the various systems perform generally better than the corresponding main-memory versions. This behaviour can be explained by considering that LUBM data sets are quite large and, consequently, they could not fit in main memory, thus causing swapping of main-memory systems.

System/Query	OQ_6	OQ_9	OQ_{11}	OQ_{15}
DLV^{DB}	4.88	5.07	5.08	5.03
Mulgara	53.12	-	-	74.23
Sesame1-Mem	256.98	318.52	320.11	268.89
Sesame1-DB	23.31	23.64	38.98	37.47
Sesame2-Mem	291.83	368.65	370.73	292.22
Sesame2-File	15.87	16.05	16.00	15.91
ARQ-Mem	309.38	329.92	331.80	312.78
ARQ-DB	17.31	343.14	407.73	154.98

Table 7.5: Memory footprint (Megabytes) of tested systems for some queries

In order to further characterize tested systems, we have measured their main-memory footprint for some queries. Table 7.5 shows the results obtained for queries OQ_6, OQ_9, OQ_{11}, and OQ_{15} run on the dataset `lubm-15`. Recall that OQ_9 and Q_{11} require inference. From the analysis of this table it clearly emerges that DLV^{DB} is the lightest system in terms of memory occupation. Moreover, it comes with no surprise that, in general, the systems requiring the lower amount of main-memory are those implementing a mass-memory based evaluation; the only exception is made by ARQ-DB in OQ_9 and OQ_{11} (requiring inference). This can be explained by considering that ARQ applies inference rules at query time and, visibly, this task is carried out in main memory. These results, coupled with the execution times previously analyzed, make DLV^{DB} the system requiring overall less time and space among the tested ones.

7.4 Testing on a combinatorial problem

In this test, we considered a combinatorial problem, we call it *FastFoods*, which checks whether a depot allocation has minimal supply costs among all depot allocations of the same cardinality. Inputs to the problem are a set of *restaurants* and a set of *depots*, each characterized by a Name and a Position (Km). The output is an alternative set of *depots*, if available. The complete encoding of this problem can be found in the Appendix A.

Note that we could test only DLV and DLV^{DB} on this problem. In fact, the encoding of *FastFoods* is heavily based on aggregate functions, especially assignment aggregates which are not supported by the other systems.

Results showing response times for increasing numbers of restaurants are illustrated in Figure 7.14(a) [7]. It clearly emerges that DLV^{DB} is much more effective than DLV in aggregating data for increasing input sizes; this can be justified by the fact that DLV^{DB} exploits DBMS aggregation functions during the grounding.

7.5 Testing on data transformation problems

We tested the capability to improve usability and efficiency of DLV^{DB} via functions for a typical real world problem, namely data transformation. Data transformation is particularly relevant, e.g. in data integration, to uniform data formats among different sources.

In particular, we considered the problem of transforming integer numbers in their binary representation. This task can be encoded both in pure datalog and in datalog with functions (see the Appendix A). We then designed a test program, named *Int2Bin*, aiming simply at transforming integers stored in an input table to binaries. We defined two variants of *Int2Bin*, one with and one without function calls. In order to measure the scalability of DLV^{DB} in this test, we considered output binary numbers having 5 to 16 bits. Obtained results are shown in Figure7.14(b).

The figure clearly shows the significant advantage of using functions in this context. In fact, the execution time of *Int2Bin* with functions is almost constant because it requires a fixed number of function calls (one for each mark to convert), independently of the number of bits. To the contrary, the standard datalog version must generate all the binary numbers in the admissible range; this explains the exponential growth of the response time.

7.6 Testing on string similarity computation

String similarity computation is an important task in several application areas. In particular, in Bioinformatics, it is essential for measuring several parameters between portions of DNA or proteins and to identify frequently repeated patterns. ASP (with some extensions) has already been exploited also in this context, see e.g. [103].

In this test, we considered the computation of the Hamming distance between pairs of strings, which is at the basis of several similarity measures. It is defined as the number of positions in which the corresponding symbols of two strings of the same length are different. This problem is inherently procedural and, even if a declarative solution for it is possible, this is quite unnatural.

We then considered the following problem, referred as HammingDistances in the following: given a set of strings compute the Hamming distance between each string pair. Note that, in

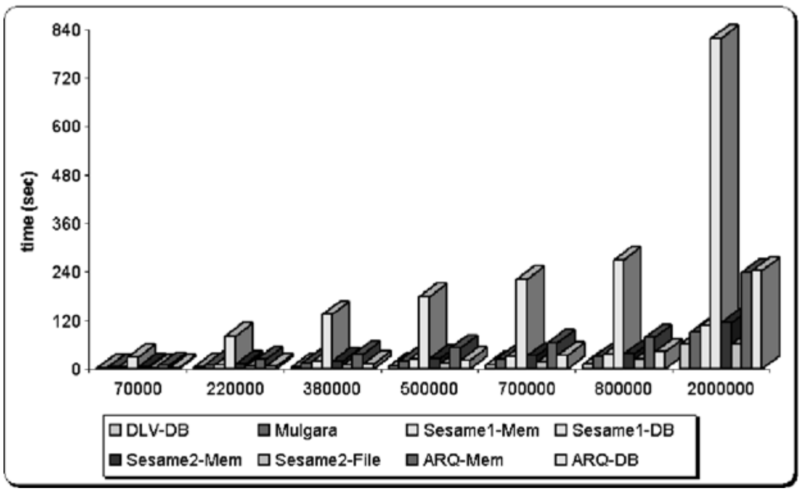
[7] We fixed the number of depots to 50.

classical ASP, in order to properly compute the hamming distance, strings must be represented as a set of pairs (POS, CHAR); to the contrary, a function-based solution can directly handle the whole string.

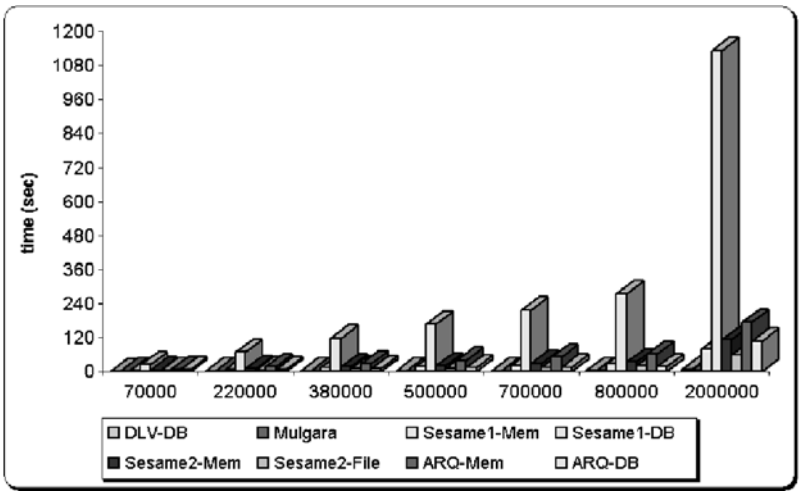
We then designed two encodings for the problem, one using functions and one not; specifically, in the former case input strings are represented as `string(ID,S)`, whereas in the latter case, strings are expressed as `string(ID,CHAR,POS)`. Note that we did not count the time for converting the strings from one format to the other in our tests. In both cases, the output has the form `hd(ID1,ID2,H)`. The complete encodings can be found in the Appendix A.

Results are shown in Figure 7.14(c) for increasing numbers of input strings. The gain provided by DLV^{DB} is similar to that we have observed in the previous test, thus confirming the advantage of using functions to solve procedural sub-tasks.

OQ_1



OQ_2



OQ_3

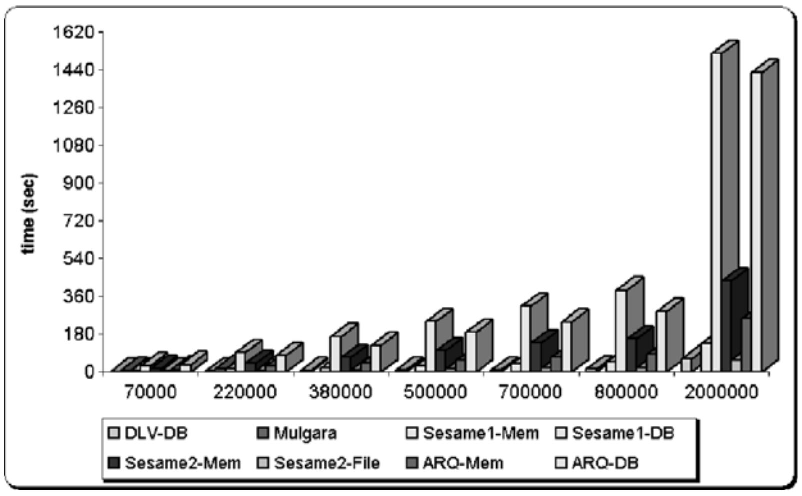
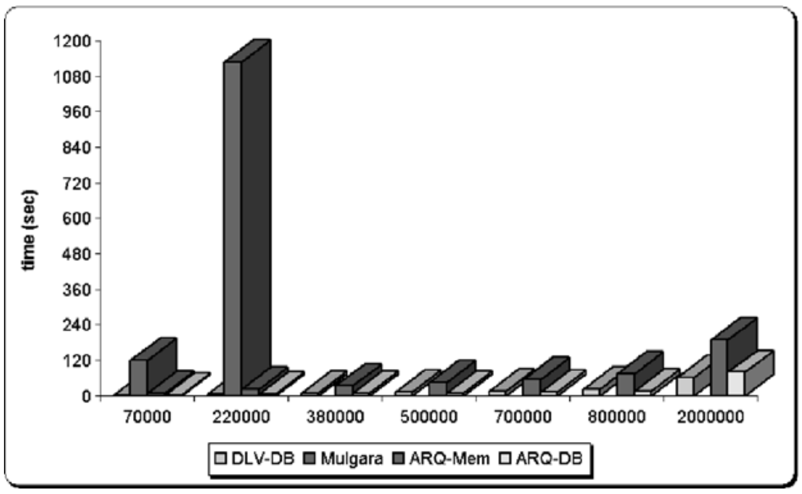


Figure 7.8: Results for queries OQ_1 - OQ_3

OQ_4



OQ_5

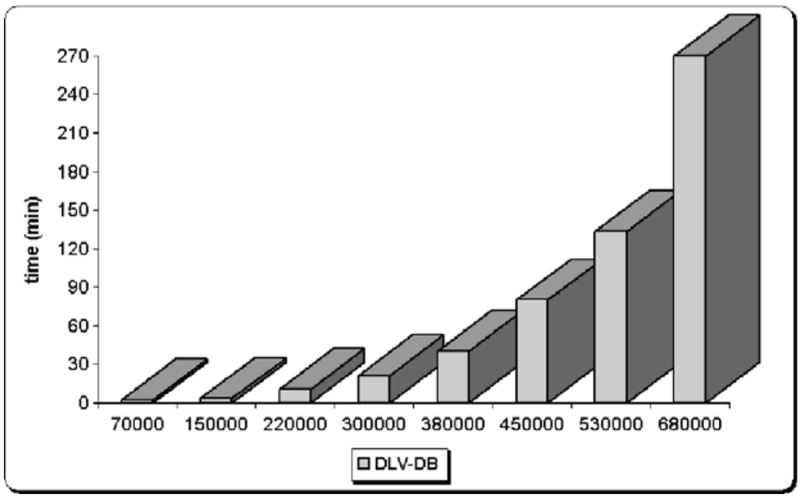
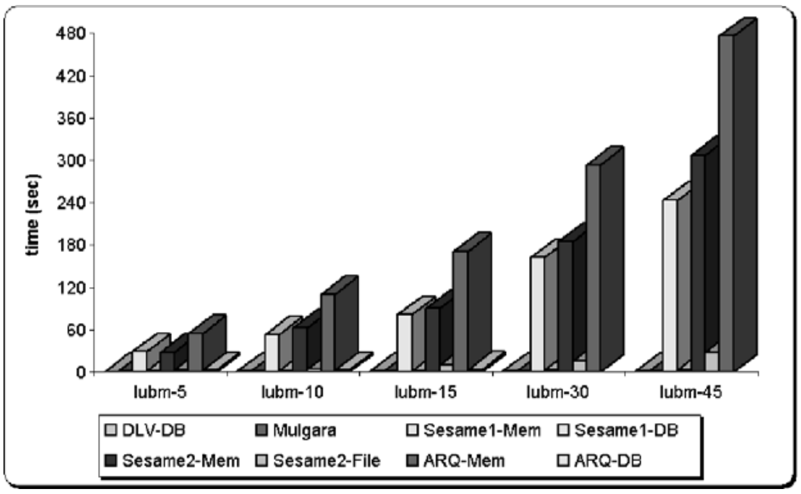
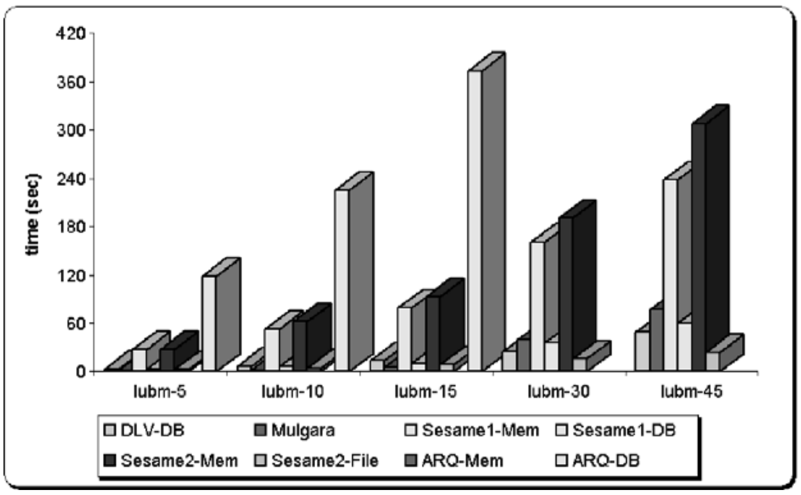


Figure 7.9: Results for queries OQ_4 and OQ_5

OQ_6



OQ_7



OQ_8

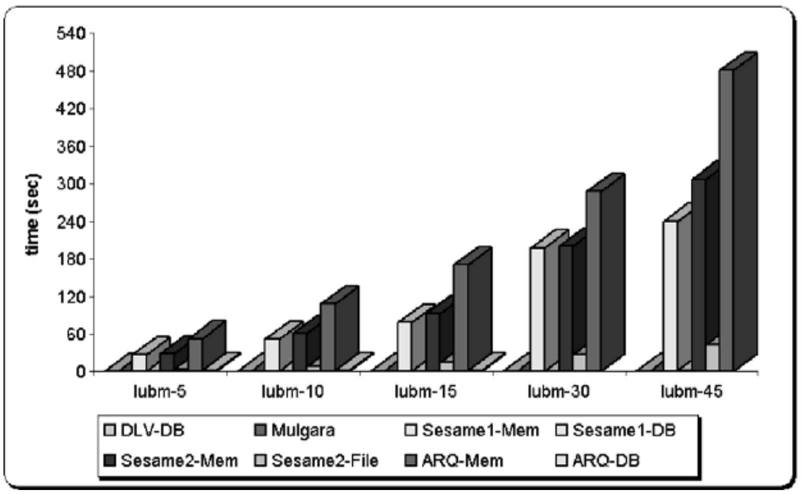
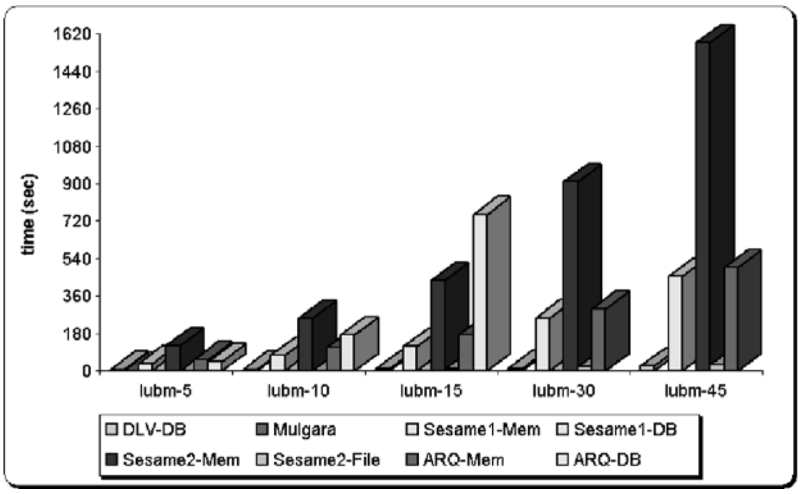
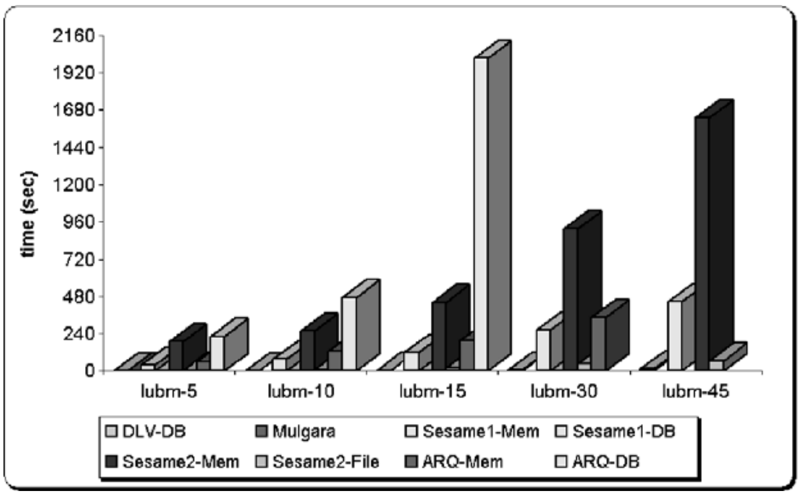


Figure 7.10: Results for queries OQ_6 - OQ_8

OQ_9



OQ_{10}



OQ_{11}

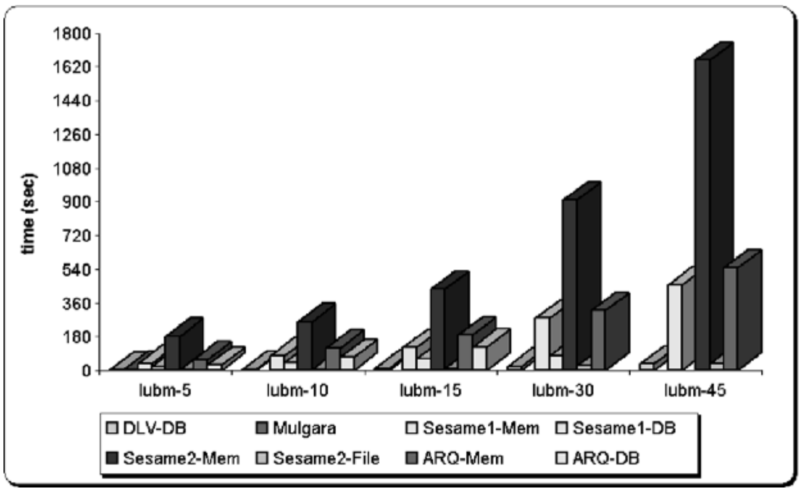
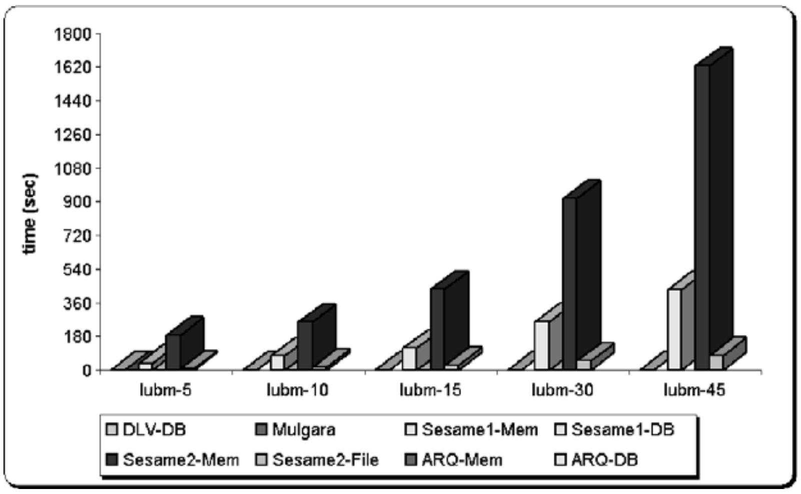
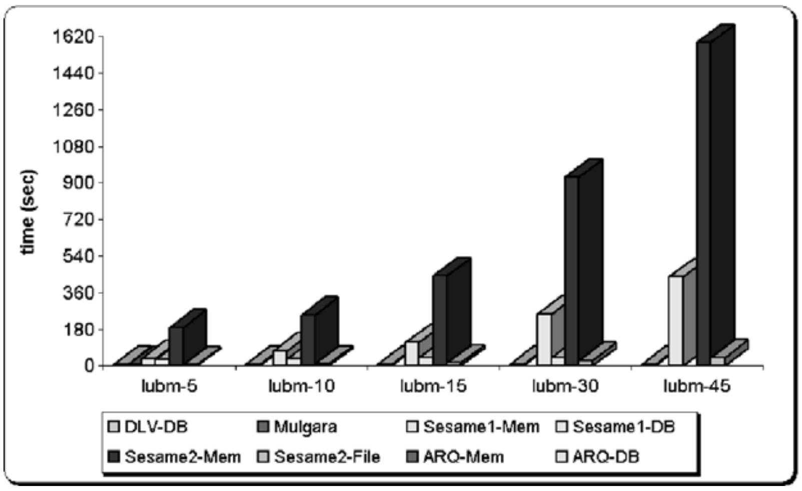


Figure 7.11: Results for queries OQ_9 - Q_{11}

OQ_{12}



OQ_{13}



OQ_{14}

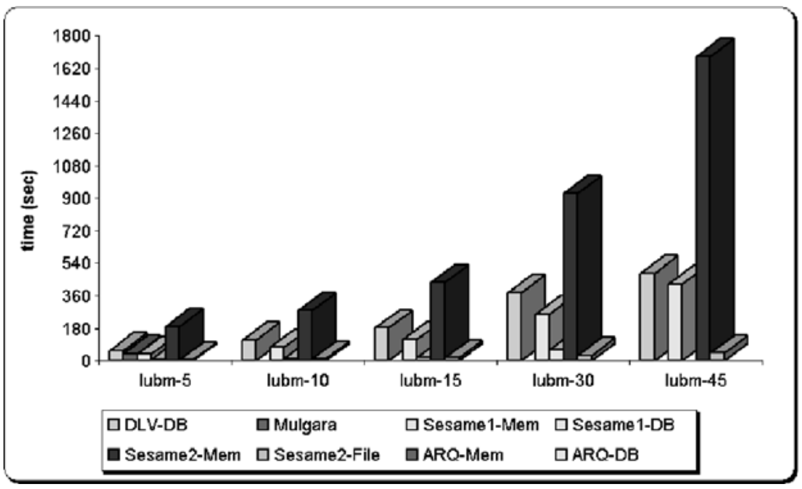
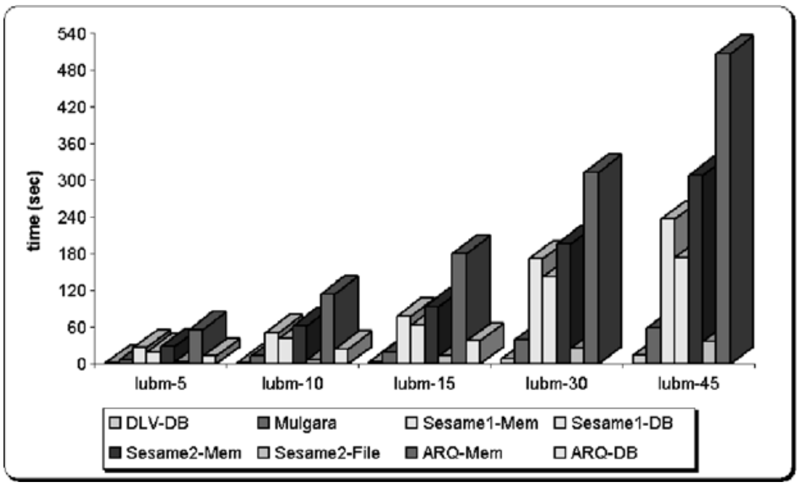
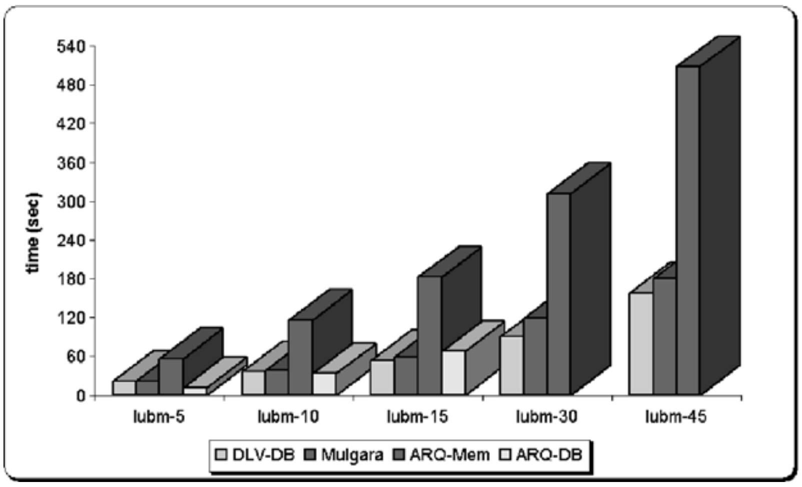


Figure 7.12: Results for queries OQ_{12} - OQ_{14}

OQ_{15}



OQ_{16}



OQ_{17}

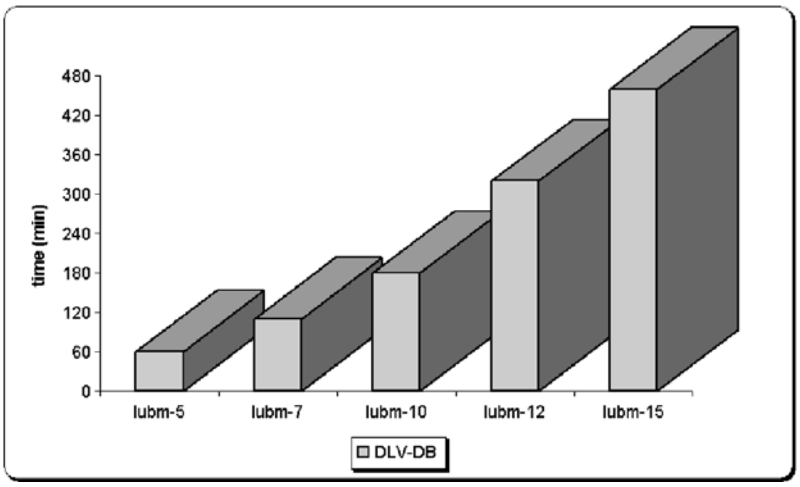
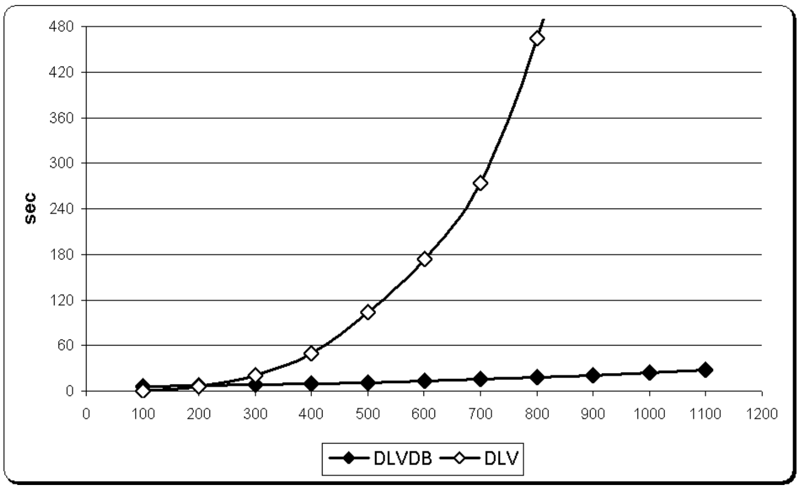
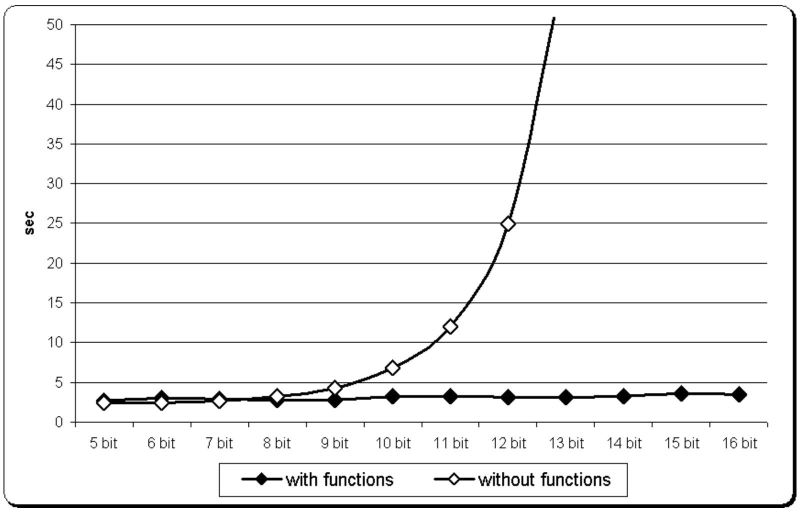


Figure 7.13: Results for queries OQ_{15} - Q_{17}

(a)



(b)



(c)

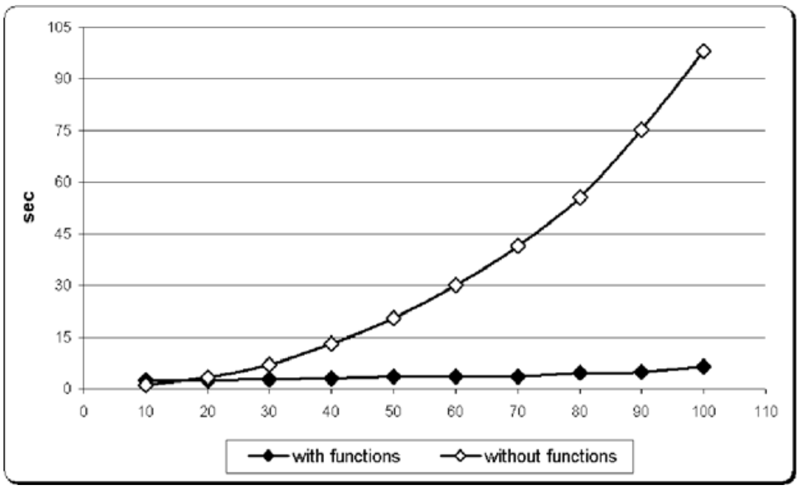


Figure 7.14: Results for (a) FastFoods; (b) Int2Bin; (c) HammingDistances.

Chapter 8

Conclusions

In this thesis we have presented DLV^{DB}, a new deductive system for reasoning on massive amounts of data. We have pointed out that DLV^{DB} is particularly suited for data intensive applications; moreover we showed that DLV^{DB} exemplifies the usage of DLP for those problems characterized by both declarative and procedural components, via the usage of external function calls.

First we have pointed out innovations and differences of our approach with respect to existing proposals by a comprehensive survey on logic-based systems and database systems. Then we have presented the overall characteristics of the proposed system, and its database oriented evaluation strategy. Finally we have presented some application scenarios possibly benefiting of DLV^{DB} and adopted for our tests whose results have been encuraging.

The main contributions of this work are the following:

- We have described the architecture of a fully fledged system enhancing in different ways the interactions between logic-based systems and DBMS*s*.
- We have introduced a novel evaluation strategy for logic programs allowing to minimize the usage of main-memory and to maximize the advantages of optimization techniques implemented in existing DBMS*s*.
- We have extended the expressiveness of DLP with external function calls, yet improving efficiency (at least for procedural sub-tasks) and knowledge-modelling power;
- We have extended expressiveness of DLP for supporting also the management of recursive data structures (lists).
- We have designed a simple and easy to use mechanism, based on ODBC, for the cooperation of the developed system with any existing DBMS.
- We have experimentally shown that DLV^{DB} effectively combines the experience in optimizing ASP programs gained within the DLV project with the well assessed data management capabilities of existing DBMS*s*. This allows to enhance performance with data intensive applications.
- We have experimentally demonstrated that the developed system outperforms existing proposals in several interesting application problems.

- We have shown how *(i)* the proposed system can be effectively exploited in complex application scenarios such as the integration of distributed, possibly inconsistent, and possibly incomplete data and *(ii)* the querying of RDF(S) ontologies.

As for future work we plan to:

- Enhance the management of distributed data in order to optimize the interaction beetwen the system and databases.
- Evaluate the possible parallelization of some tasks in the evaluation process, mainly for the elaboration of distributed data.
- Introduce data typing features, in order to allow the handling of complex domains and to pone the way to the manipulation of objects.

Appendix A

Encodings

A.1 Testing on normal stratified programs with recursion

A.1.1 Encodings of Reachability

DLV^{DB} and DLV^{IO} :

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

DLV^{DB} and DLV^{IO} allow to add the queries directly to the datalog program. Queries used for DLV^{DB} and DLV^{IO} are: $\mathcal{Q}_0$=$reachable(X, Y)?$, $\mathcal{Q}_1$=$reachable(b1, Y)?$ and $\mathcal{Q}_2$=$reachable(b1, b2)?$, where $b1$ and $b2$ represent constant values.

SQLServer and DB2: Both SQLServer and DB2 require two different encodings depending on the kind of underlying data structure (cyclic or acyclic); in fact, in case of cyclic data structures, they require explicit mechanisms in the query to avoid infinite loops. Clearly such verifications can worsen the performance of the computation. As a consequence, in order to guarantee fairness in our benchmarks, we used the encoding with these verifications only for queries on cyclic data structure.

SQLServer and DB2 for acyclic data structures:

```
WITH reachable($att_1$, $att_2$) AS (
        SELECT $att_1$, $att_2$
        FROM edge
    UNION ALL
        SELECT edge.$att_1$, reachable.$att_2$
        FROM edge, reachable
        WHERE edge.$att_2$ = reachable.$att_1$ )
    SELECT $att_1$, $att_2$
    FROM reachable
    ORDER BY $att_1$, $att_2$;
```

SQLServer for cyclic data structures: The way to avoid infinite loops on cyclic data structures consists in limiting the length of the discovered paths; this requires both to compute path lengths and to limit the maximum number of recursive iterations to be carried out.

```
WITH reachable($att_1, att_2, path\_length$) AS (
        SELECT $att_1, att_2$, 1
        FROM edge
    UNION ALL
        SELECT edge.$att_1$, reachable.$att_2$, **reachable.**$path\_length$ **+ 1**
        FROM edge, reachable
        WHERE edge.$att_2$ = reachable.$att_1$
                **AND reachable.**$path\_length < \#node$**)**
    SELECT $att_1, att_2$, MIN($path\_length$)
    FROM reachable
    **GROUP BY** $att_1, att_2$
    **OPTION** (MAXRECURSION = $\#node$)
```

Here, in **Bold** text style we have highlighted the additional conditions required w.r.t. the basic case. Moreover, the parameter $\#node$ is used to limit the maximum length of derived paths; in order to guarantee fairness, for each test, we have set $\#node$ to the number of nodes actually present in the exploited graph instance.

DB2 for cyclic data structures: DB2 requires the limitation on the path lengths, but not the one on the maximum number of recursive iterations.

```
WITH reachable($att_1, att_2, path\_length$) AS (
        SELECT $att_1, att_2$, 1
        FROM edge
    UNION ALL
        SELECT edge.$att_1$, reachable.$att_2$, **reachable.**$path\_length$ **+ 1**
        FROM edge, reachable
        WHERE edge.$att_2$ = reachable.$att_1$
                **AND reachable.**$path\_length < \#node$**)**
    SELECT $att_1, att_2$, MIN($path\_length$)
    FROM reachable
    **GROUP BY** $att_1, att_2$
```

All the SQL99 statements above solve the unbound query $\mathcal{Q}_0$; customizing them for queries with one (resp., two) parameter bound amounts just in adding one (resp., two) $WHERE$ condition in the main selection statement.

LDL++:

$$reachable(X, Y) \leftarrow edge(X, Y).$$
$$reachable(X, Y) \leftarrow edge(X, Z),\ reachable(Z, Y).$$

The substantial difference between LDL++ and DLV^{DB} encodings resides in query definition. In particular, LDL++ exploits query forms (also called *export*), which are generic queries specifying to the compiler which arguments will be given in input and which ones are expected as output. For instance, if the program file contains the expression: $export\ reachable(\$X, Y)$, then it will be possible to execute queries of the form: $reachable(\text{‘}Rome\text{’}, X)$. The bindings given in an export are used by the compiler to optimize data accesses.

Then, LDL++ has been tested with the queries: $\mathcal{Q}_0$=$export\ reachable(X, Y)$, $\mathcal{Q}_1$=$export\ reachable(\$X, Y)$ and $\mathcal{Q}_2$=$export\ reachable(\$X, \$Y)$.

XSB: XSB requires *memoization* in order to ensure that the program terminates; it consists in declaring some predicates as *tabled*. The XSB encoding is the following:

$: -table\ reachable/2.$

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

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

The used queries are the same as those presented for DLV^{DB}.

Smodels: As for Smodels, we had to slightly modify the standard encoding in order to respect the *domain restriction* constraint required by the instantiation module. This module is a separate application, called *lparse*, which preprocesses the programs evaluated by Smodels.

The Smodels encoding is the following:

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

$reachable(X,\, Y)\ \ \text{:-}\ edge(X, Z),\ reachable(Z,\, Y),\ vertex(Y).$

where $vertex(Y)$ has been added to restrict the domain of the variable Y. The used queries are the same as those presented for DLV^{DB}.

A.1.2 Encodings of Same Generation

DLV^{DB} and DLV^{IO} :

$samegen(X,\, Y)\ \ \text{:-}\ parent(P, X),\ parent(P,\, Y).$

$samegen(X,\, Y)\ \ \text{:-}\ parent(P_1, X),\ parent(P_2,\, Y),\ samegen(P_1, P_2).$

with the queries $\mathcal{Q}_0$=$samegen(X, Y)$?, $\mathcal{Q}_1$=$samegen(b1, Y)$? and $\mathcal{Q}_2$=$samegen(b1, b2)$?.

SQLServer and DB2:

```
WITH samegen($att_1$, $att_2$) AS (
        SELECT $\mathcal{P}_1.att_2$, $\mathcal{P}_2.att_2$
        FROM parent $\mathcal{P}_1$, parent $\mathcal{P}_2$
        WHERE $\mathcal{P}_1.att_1$ = $\mathcal{P}_2.att_1$
    UNION ALL
        SELECT $\mathcal{P}_1.att_2$, $\mathcal{P}_2.att_2$
        FROM parent $\mathcal{P}_1$, parent $\mathcal{P}_2$, samegen $S$
        WHERE $\mathcal{P}_1.att_1$ = $S.att_1$ AND $\mathcal{P}_2.att_1$ = $S.att_2$ )
```

SELECT att_1, att_2
FROM samegen
ORDER BY att_1, att_2;

The statement above solves the unbound query $\mathcal{Q}_0$. Customizing it for query $\mathcal{Q}_1$ (resp., $\mathcal{Q}_2$) amounts just in adding one (resp., two) $WHERE$ condition in the main selection statement.

LDL++:

$$samegen(X, Y) \leftarrow parent(P, X),\ parent(P, Y).$$
$$samegen(X, Y) \leftarrow parent(P_1, X),\ parent(P_2, Y),\ samegen(P_1, P_2).$$

with queries $\mathcal{Q}_0$=$export\ samegen(X, Y)$, $\mathcal{Q}_1$=$export\ samegen(\$X, Y)$ and $\mathcal{Q}_2$ =$export\ samegen(\$X, \$Y)$.

XSB:

$$:-table\ samegen/2$$
$$samegen(X, Y)\ :\text{-}\ parent(P, X),\ parent(P, Y).$$
$$samegen(X, Y)\ :\text{-}\ parent(P_1, X),\ parent(P_2, Y),\ samegen(P_1, P_2).$$

with the same queries as the ones shown for DLV^{DB}.

Smodels:

$$samegen(X, Y)\ :\text{-}\ parent(P, X),\ parent(P, Y).$$
$$samegen(X, Y)\ :\text{-}\ parent(P_1, X),\ parent(P_2, Y),\ samegen(P_1, P_2).$$

with the same queries as the ones shown for DLV^{DB}.

A.2 Testing on a real data integration setting

A.2.1 Encodings for query IQ_1

The encodings of query IQ_1 exploited in our tests are the following.

DLV^{DB} and DLV encoding of query IQ_1

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
          dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
          exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
          exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
          exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2.

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

q1(ED,RN) :- course(EC,ED), exam_record_inf(RN,EC,FN,LN,GR,DT,AY).

q1(ED,RN)?
```

GnT2, ClaspD and Cmodels encoding of query IQ_1 with Lparse

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2.

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

q1(ED,RN) :- course(EC,ED), exam_record_inf(RN,EC,FN,LN,GR,DT,AY),
        exam_record_infD(RN,EC,FN,LN,GR,DT,AY).
```

`smodels` encoding of query IQ_1 with Lparse

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).
```

```
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

q1(ED,RN) :- course(EC,ED), exam_record_inf(RN,EC,FN,LN,GR,DT,AY),
        exam_record_infD(RN,EC,FN,LN,GR,DT,AY).
```

GnT2, ClaspD, Cmodels and `smodels` encoding of query IQ_1 with GrinGo

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
```

```
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

q1(ED,RN) :- course(EC,ED), exam_record_inf(RN,EC,FN,LN,GR,DT,AY),
        exam_record_infD(RN,EC,FN,LN,GR,DT,AY).
```

A.2.2 Encodings for query IQ_2

The encodings of query Q_2 exploited in our tests are the following.

DLV^{DB} and DLV encoding of query IQ_2

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2.

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_professori(PC,FN,LN).

teaching(EC,FN,LN,AY) :- teachingD(EC,FN,LN,AY), not teachingC(EC,FN,LN,AY).

professorD(FN,LN) :- professore(PC,FN,LN,CI).
professorD(FN,LN) :- dati_professori(PC,FN,LN).
professorD(FN,LN) :- professorWeb__professor(PC,LN,FN,TI,HP,PN,FN,EM,AD).

professor(FN,LN) :- professorD(FN,LN), not professorC(FN,LN).

q2(FN,LN) :- teaching(EC,FN,LN,AY).
q2(FN,LN) :- professor(FN,LN).
```

```
q2(FN,LN) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY).

q2(FN,LN)?
```

GnT2, ClaspD and Cmodels encoding of query IQ_2 with Lparse

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2.

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_professori(PC,FN,LN).

teaching(EC,FN,LN,AY) :- teachingD(EC,FN,LN,AY), not teachingC(EC,FN,LN,AY).

professorD(FN,LN) :- professore(PC,FN,LN,CI).
professorD(FN,LN) :- dati_professori(PC,FN,LN).
professorD(FN,LN) :- professorWeb__professor(PC,LN,FN,TI,HP,PN,FN,EM,AD).

professor(FN,LN) :- professorD(FN,LN), not professorC(FN,LN).

q2(FN,LN) :- teaching(EC,FN,LN,AY).
q2(FN,LN) :- professor(FN,LN).
q2(FN,LN) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY),
        exam_record_infD(RN,EC,FN,LN,GR,DT,AY).
```

`smodels` encoding of query IQ_2 with Lparse

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_professori(PC,FN,LN).

teaching(EC,FN,LN,AY) :- teachingD(EC,FN,LN,AY), not teachingC(EC,FN,LN,AY).

professorD(FN,LN) :- professore(PC,FN,LN,CI).
professorD(FN,LN) :- dati_professori(PC,FN,LN).
professorD(FN,LN) :- professorWeb__professor(PC,LN,FN,TI,HP,PN,FN,EM,AD).

professor(FN,LN) :- professorD(FN,LN), not professorC(FN,LN).

q2(FN,LN) :- teaching(EC,FN,LN,AY).
q2(FN,LN) :- professor(FN,LN).
q2(FN,LN) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY),
        exam_record_infD(RN,EC,FN,LN,GR,DT,AY).
```

GnT2, ClaspD, Cmodels and `smodels` **encoding of query** IQ_2 **with GrinGo**

```
exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
            dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN).

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
            not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
            not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
            not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
            not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
```

```
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_professori(PC,FN,LN).

teaching(EC,FN,LN,AY) :- teachingD(EC,FN,LN,AY), not teachingC(EC,FN,LN,AY).

professorD(FN,LN) :- professore(PC,FN,LN,CI).
professorD(FN,LN) :- dati_professori(PC,FN,LN).
professorD(FN,LN) :- professorWeb__professor(PC,LN,FN,TI,HP,PN,FN,EM,AD).

professor(FN,LN) :- professorD(FN,LN), not professorC(FN,LN).

q2(FN,LN) :- teaching(EC,FN,LN,AY).
q2(FN,LN) :- professor(FN,LN).
q2(FN,LN) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY).
```

A.2.3 Encodings for query IQ_3

The encodings of query IQ_3 exploited in our tests are the following.

DLV^{DB} and DLV encoding of query IQ_3

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
        studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                        AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).

studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2),LN1 != LN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2.
```

```
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2.

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
        not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_recordD(RN,EC,GR,DT,PC) :- dati_esami(RN,CI,EC,DT,GR,RE,PC), GR > k [1].

exam_recordC(RN,EC,GR1,DT1,AY1) ∨ exam_recordC(RN,EC,GR2,DT2,AY2) :-
        exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), GR1 != GR2.
exam_recordC(RN,EC,GR1,DT1,AY1) ∨ exam_recordC(RN,EC,GR2,DT2,AY2) :-
        exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), DT1 != DT2.
exam_recordC(RN,EC,GR1,DT1,AY1) ∨ exam_recordC(RN,EC,GR2,DT2,AY2) :-
        exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), AY1 != AY2.

exam_record(RN,EC,GR,DT,AY) :- exam_recordD(RN,EC,GR,DT,AY),
        not exam_recordC(RN,EC,GR,DT,AY).

pstudent(RN,LN) :- student(RN,FN,LN,AC,AD,TN,DE).

pexam_record(RN,EC) :- exam_record(RN,EC,GR,DT,AY).

q3(LN1,LN2) :- pstudent(RN1,LN1), pstudent(RN2,LN2), RN1 != RN2,
        pexam_record(RN1,EC), pexam_record(RN2,EC).

q3(LN1,LN2)?
```

GnT2, ClaspD and Cmodels encoding of query IQ_3 with Lparse

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
        studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                    AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).

studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2.
```

[1] Here, the condition GR $> k$ has been introduced to leverage the amount of data accessed by the query, in order to study the scalability of tested systems. Specifically, k has been varied between 0 and 30; k=0 takes all exams, whereas k=30 takes only those exams with marks above 30 (corresponding to 10% of the total amount of exams).

```
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2.

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
        not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_recordD(RN,EC,GR,DT,PC) :- dati_esami(RN,CI,EC,DT,GR,RE,PC), GR > k 1.

exam_recordC(RN,EC,GR1,DT1,AY1) | exam_recordC(RN,EC,GR2,DT2,AY2) :-
        exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), GR1 != GR2.
exam_recordC(RN,EC,GR1,DT1,AY1) | exam_recordC(RN,EC,GR2,DT2,AY2) :-
        exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), DT1 != DT2.
exam_recordC(RN,EC,GR1,DT1,AY1) | exam_recordC(RN,EC,GR2,DT2,AY2) :-
        exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), AY1 != AY2.

exam_record(RN,EC,GR,DT,AY) :- exam_recordD(RN,EC,GR,DT,AY),
        not exam_recordC(RN,EC,GR,DT,AY).

pstudent(RN,LN) :- student(RN,FN,LN,AC,AD,TN,DE), studentD(RN,FN,LN,AC,AD,TN,DE).

pexam_record(RN,EC) :- exam_record(RN,EC,GR,DT,AY), exam_recordD(RN,EC,GR,DT,AY).

q3(LN1,LN2) :- pstudent(RN1,LN1), studentD(RN1,FN1,LN1,AC1,AD1,TN1,DE1),
        pstudent(RN2,LN2), studentD(RN2,FN2,LN2,AC2,AD2,TN2,DE2), RN1 != RN2,
        pexam_record(RN1,EC), exam_recordD(RN1,EC,GR1,DT1,AY1),
        pexam_record(RN2,EC), exam_recordD(RN2,EC,GR2,DT2,AY2).
```

`smodels` encoding of query IQ_3 with Lparse

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
        studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                    AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).

studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
        not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_recordD(RN,EC,GR,DT,PC) :- dati_esami(RN,CI,EC,DT,GR,RE,PC), GR > k [1].

exam_recordC(RN,EC,GR1,DT1,AY1) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), GR1 != GR2,
        not exam_recordC(RN,EC,GR2,DT2,AY2).
exam_recordC(RN,EC,GR2,DT2,AY2) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), GR1 != GR2,
        not exam_recordC(RN,EC,GR1,DT1,AY1).
exam_recordC(RN,EC,GR1,DT1,AY1) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), DT1 != DT2,
        not exam_recordC(RN,EC,GR2,DT2,AY2).
exam_recordC(RN,EC,GR2,DT2,AY2) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), DT1 != DT2,
        not exam_recordC(RN,EC,GR1,DT1,AY1).
exam_recordC(RN,EC,GR1,DT1,AY1) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), AY1 != AY2,
        not exam_recordC(RN,EC,GR2,DT2,AY2).
exam_recordC(RN,EC,GR2,DT2,AY2) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), AY1 != AY2,
        not exam_recordC(RN,EC,GR1,DT1,AY1).

exam_record(RN,EC,GR,DT,AY) :- exam_recordD(RN,EC,GR,DT,AY),
        not exam_recordC(RN,EC,GR,DT,AY).

pstudent(RN,LN) :- student(RN,FN,LN,AC,AD,TN,DE), studentD(RN,FN,LN,AC,AD,TN,DE).

pexam_record(RN,EC) :- exam_record(RN,EC,GR,DT,AY), exam_recordD(RN,EC,GR,DT,AY).

q3(LN1,LN2) :- pstudent(RN1,LN1), studentD(RN1,FN1,LN1,AC1,AD1,TN1,DE1),
        pstudent(RN2,LN2), studentD(RN2,FN2,LN2,AC2,AD2,TN2,DE2), RN1 != RN2,
        pexam_record(RN1,EC), exam_recordD(RN1,EC,GR1,DT1,AY1),
        pexam_record(RN2,EC), exam_recordD(RN2,EC,GR2,DT2,AY2).
```

GnT2, ClaspD, Cmodels and `smodels` encoding of query IQ_3 with GrinGo

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
            studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                   AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).

studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
```

```
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
        not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_recordD(RN,EC,GR,DT,PC) :- dati_esami(RN,CI,EC,DT,GR,RE,PC), GR > k 1.

exam_recordC(RN,EC,GR1,DT1,AY1) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), GR1 != GR2,
        not exam_recordC(RN,EC,GR2,DT2,AY2).
exam_recordC(RN,EC,GR2,DT2,AY2) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), GR1 != GR2,
        not exam_recordC(RN,EC,GR1,DT1,AY1).
exam_recordC(RN,EC,GR1,DT1,AY1) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), DT1 != DT2,
        not exam_recordC(RN,EC,GR2,DT2,AY2).
exam_recordC(RN,EC,GR2,DT2,AY2) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), DT1 != DT2,
        not exam_recordC(RN,EC,GR1,DT1,AY1).
exam_recordC(RN,EC,GR1,DT1,AY1) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), AY1 != AY2,
        not exam_recordC(RN,EC,GR2,DT2,AY2).
exam_recordC(RN,EC,GR2,DT2,AY2) :- exam_recordD(RN,EC,GR1,DT1,AY1),
        exam_recordD(RN,EC,GR2,DT2,AY2), AY1 != AY2,
        not exam_recordC(RN,EC,GR1,DT1,AY1).

exam_record(RN,EC,GR,DT,AY) :- exam_recordD(RN,EC,GR,DT,AY),
        not exam_recordC(RN,EC,GR,DT,AY).

pstudent(RN,LN) :- student(RN,FN,LN,AC,AD,TN,DE).

pexam_record(RN,EC) :- exam_record(RN,EC,GR,DT,AY).

q3(LN1,LN2) :- pstudent(RN1,LN1), pstudent(RN2,LN2), RN1 != RN2,
        pexam_record(RN1,EC), pexam_record(RN2,EC).
```

A.2.4 Encodings for query IQ_4

The encodings of query Q_4 exploited in our tests are the following.

DLV^{DB} and DLV encoding of query IQ_4

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
        studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                        AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).
```

```
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2),LN1 != LN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) ∨ studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2.

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
        not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN), GR > k [1].

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) ∨ exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2.

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).

courseC(EC,ED1) ∨ courseC(EC,ED2) :- courseD(EC,ED1), courseD(EC,ED2), ED1 != ED2.

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_professori(PC,FN,LN).

pstudent(RN) :- student(RN,FN,LN,AC,AD,TN,DE).

pexam_record_inf(RN,EC) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY).
```

```
pcourse(EC) :- course(EC,ED).

pteachingD(EC,FN) :- teachingD(EC,FN,LN,AY).

q4(RN1,EC1,RN2,EC2) :- pstudent(RN1), pstudent(RN2), RN1 != RN2,
        pexam_record_inf(RN1,EC1), pexam_record_inf(RN2,EC2),
        pcourse(EC1), pcourse(EC2), EC1 != EC2,
        pteachingD(EC1,FN), pteachingD(EC2,FN).

q4(RN1,EC1,RN2,EC2)?
```

GnT2, ClaspD and Cmodels encoding of query IQ_4 with Lparse

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
        studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).

studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2),LN1 != LN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2.
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) | studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :-
        studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2.

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
        not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN), GR > k 1.

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
        exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2.
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) | exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :-
```

```
          exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
          exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2.

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
          not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).

courseC(EC,ED1) | courseC(EC,ED2) :- courseD(EC,ED1), courseD(EC,ED2), ED1 != ED2.

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
          dati_professori(PC,FN,LN).

pstudent(RN) :- student(RN,FN,LN,AC,AD,TN,DE), studentD(RN,FN,LN,AC,AD,TN,DE).

pexam_record_inf(RN,EC) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY),
          exam_record_infD(RN,EC,FN,LN,GR,DT,AY).

pcourse(EC) :- course(EC,ED), courseD(EC,ED).

pteachingD(EC,FN) :- teachingD(EC,FN,LN,AY).

q4(RN1,EC1,RN2,EC2) :- pstudent(RN1), pstudent(RN2), RN1 != RN2,
          pexam_record_inf(RN1,EC1), pexam_record_inf(RN2,EC2),
          pcourse(EC1), pcourse(EC2), EC1 != EC2,
          pteachingD(EC1,FN), pteachingD(EC2,FN),
          exam_record_infD(RN1,EC3,FN1,LN1,GR1,DT1,AY1),
          exam_record_infD(RN2,EC4,FN2,LN2,GR2,DT2,AY2).
```

`smodels` encoding of query IQ_4 with Lparse

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
          studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                        AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).

studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
          studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
          not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
          studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
          not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
          studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
          not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
          studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
          not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
```

```
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
        not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN), GR > k 1.

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
        exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
        not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
        not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).
```

```
courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).

courseC(EC,ED1) :- courseD(EC,ED1), courseD(EC,ED2), ED1 != ED2, not courseC(EC,ED2).
courseC(EC,ED2) :- courseD(EC,ED1), courseD(EC,ED2), ED1 != ED2, not courseC(EC,ED1).

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_professori(PC,FN,LN).

pstudent(RN) :- student(RN,FN,LN,AC,AD,TN,DE), studentD(RN,FN,LN,AC,AD,TN,DE).

pexam_record_inf(RN,EC) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY),
        exam_record_infD(RN,EC,FN,LN,GR,DT,AY).

pcourse(EC) :- course(EC,ED), courseD(EC,ED).

pteachingD(EC,FN) :- teachingD(EC,FN,LN,AY).

q4(RN1,EC1,RN2,EC2) :- pstudent(RN1), pstudent(RN2), RN1 != RN2,
        pexam_record_inf(RN1,EC1), pexam_record_inf(RN2,EC2),
        pcourse(EC1), pcourse(EC2), EC1 != EC2,
        pteachingD(EC1,FN), pteachingD(EC2,FN),
        exam_record_infD(RN1,EC3,FN1,LN1,GR1,DT1,AY1),
        exam_record_infD(RN2,EC4,FN2,LN2,GR2,DT2,AY2).
```

GnT2, ClaspD, Cmodels and `smodels` encoding of query IQ_4 with GrinGo

```
studentD(RN,FN,LN,AC2,AD2,TN1,DE) :- diploma_maturita(DG,DE),
        studente(RN,LN,FN,BD,BC,BR,AD1,AN1,ZC1,AC1,AR1,TP1,TN1,
                                        AD2,AN2,ZC2,AC2,AR2,TP2,TN2,CF,DG,SD).

studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), FN1 != FN2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), LN1 != LN2,
        not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
        studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
        not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AC1 != AC2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), AD1 != AD2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), TN1 != TN2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).
studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
            not studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2).
studentC(RN,FN2,LN2,AC2,AD2,TN2,DE2) :- studentD(RN,FN1,LN1,AC1,AD1,TN1,DE1),
            studentD(RN,FN2,LN2,AC2,AD2,TN2,DE2), DE1 != DE2,
            not studentC(RN,FN1,LN1,AC1,AD1,TN1,DE1).

student(RN,FN,LN,AC,AD,TN,DE) :- studentD(RN,FN,LN,AC,AD,TN,DE),
            not studentC(RN,FN,LN,AC,AD,TN,DE).

exam_record_infD(RN,EC,FN,LN,GR,DT,AY) :- affidamenti_ing_informatica(EC,PC,AY),
            dati_esami(RN,CI,EC,DT,GR,RE,PC), dati_professori(PC,FN,LN), GR > k [1].

exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
            not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), GR1 != GR2,
            not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
            not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), DT1 != DT2,
            not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).
exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
            not exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2).
exam_record_infC(RN,EC,FN,LN,GR2,DT2,AY2) :- exam_record_infD(RN,EC,FN,LN,GR1,DT1,AY1),
            exam_record_infD(RN,EC,FN,LN,GR2,DT2,AY2), AY1 != AY2,
            not exam_record_infC(RN,EC,FN,LN,GR1,DT1,AY1).

exam_record_inf(RN,EC,FN,LN,GR,DT,AY) :- exam_record_infD(RN,EC,FN,LN,GR,DT,AY),
            not exam_record_infC(RN,EC,FN,LN,GR,DT,AY).

courseD(EC,ED) :- esame(FC,EC,ED,AC).
courseD(EC,ED) :- esame_diploma(EC,ED).
```

```
courseC(EC,ED1) :- courseD(EC,ED1), courseD(EC,ED2), ED1 != ED2, not courseC(EC,ED2).
courseC(EC,ED2) :- courseD(EC,ED1), courseD(EC,ED2), ED1 != ED2, not courseC(EC,ED1).

course(EC,ED) :- courseD(EC,ED), not courseC(EC,ED).

teachingD(EC,FN,LN,AY) :- affidamenti_ing_informatica(EC,PC,AY),
        dati_professori(PC,FN,LN).

pstudent(RN) :- student(RN,FN,LN,AC,AD,TN,DE).

pexam_record_inf(RN,EC) :- exam_record_inf(RN,EC,FN,LN,GR,DT,AY).

pcourse(EC) :- course(EC,ED).

pteachingD(EC,FN) :- teachingD(EC,FN,LN,AY).

q4(RN1,EC1,RN2,EC2) :- pstudent(RN1), pstudent(RN2), RN1 != RN2,
        pexam_record_inf(RN1,EC1), pexam_record_inf(RN2,EC2),
        pcourse(EC1), pcourse(EC2), EC1 != EC2,
        pteachingD(EC1,FN), pteachingD(EC2,FN).
```

A.3 Testing on querying of DBLP ontology

A.3.1 Encodings for query OQ_1

The encodings of query OQ_1 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_1

```
result1(Name, Art) :-
    triple(Art, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
        "http://sw.deri.org/˜aharth/2004/07/dblp/dblp.owl#Article"),
    triple(Pers, "http://xmlns.com/foaf/0.1/name", Name),
    triple(Art, "http://purl.org/dc/elements/1.1/creator", Pers).
```

MULGARA encoding of query OQ_1

```
SELECT $Name $Art
FROM <rmi://localhost/server1#dblp>
WHERE $Art <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://sw.deri.org/˜aharth/2004/07/dblp/dblp.owl#Article>
AND $Art <http://purl.org/dc/elements/1.1/creator> $Pers
AND $Pers <http://xmlns.com/foaf/0.1/name> $Name;
```

SESAME1 encoding of query OQ_1

```
SELECT DISTINCT Name, Art
FROM {Art} <http://purl.org/dc/elements/1.1/creator> {Pers},
    {Art} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type},
    {Pers} <http://xmlns.com/foaf/0.1/name> {Name}
WHERE Type = <http://sw.deri.org/˜aharth/2004/07/dblp/dblp.owl#Article>
```

ARQ encoding of query OQ_1

```
SELECT DISTINCT ?Name ?Art
WHERE {
    ?Art <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Article>.
    ?Art <http://purl.org/dc/elements/1.1/creator> ?Pers.
    ?Pers <http://xmlns.com/foaf/0.1/name> ?Name.
}
```

A.3.2 Encodings for query OQ_2

The encodings of query OQ_2 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_2

```
result2(Name) :-
    triple(Art, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Article"),
    triple(Pers, "http://xmlns.com/foaf/0.1/name", Name),
    triple(Art, "http://purl.org/dc/elements/1.1/creator", Pers),
    triple(Art, "http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#year", "2000").
```

MULGARA encoding of query OQ_2

```
SELECT $Name
FROM <rmi://localhost/server1#dblp>
WHERE $Art <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Article>
AND $Art <http://purl.org/dc/elements/1.1/creator> $Pers
AND $Pers <http://xmlns.com/foaf/0.1/name> $Name
AND $Art <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#year> '2000';
```

SESAME1 encoding of query OQ_2

```
SELECT DISTINCT Name
FROM {Art} <http://purl.org/dc/elements/1.1/creator> {Pers},
    {Art} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type},
    {Pers} <http://xmlns.com/foaf/0.1/name> {Name},
    {Art} <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#year> {Year}
WHERE Type = <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Article>
AND label(Year) = "2000"
```

ARQ encoding of query OQ_2

$SELECT\ DISTINCT\ ?Name$
$WHERE$ {
  $?Art$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Article>.
  $?Art$ <http://purl.org/dc/elements/1.1/creator> $?Pers.$
  $?Pers$ <http://xmlns.com/foaf/0.1/name> $?Name.$
  $?Art$ <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#year> $?Year.$
  $FILTER$ ($?Year =$ ‘2000’)
}

A.3.3 Encodings for query OQ_3

The encodings of query OQ_3 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_3

$result3(Name, Doc)$:-
  $triple(Doc,$ “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”,
      “http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Document”$)$,
  $triple(Doc,$ “http://purl.org/dc/elements/1.1/creator”, $Pers)$,
  $triple(Pers,$ “http://xmlns.com/foaf/0.1/name”, $Name)$.

MULGARA encoding of query OQ_3

$SELECT\ \$Name\ \$Doc$
$FROM$ <rmi://localhost/server1#dblp>
$WHERE\ \$Doc$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Document>
$AND\ \$Doc$ <http://purl.org/dc/elements/1.1/creator> $\$Pers$
$AND\ \$Pers$ <http://xmlns.com/foaf/0.1/name> $\$Name;$

SESAME1 encoding of query OQ_3

$SELECT\ DISTINCT\ Name,\ Doc$
$FROM$ $\{Doc\}$ <http://purl.org/dc/elements/1.1/creator> $\{Pers\}$,
  $\{Doc\}$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> $\{Type\}$,
  $\{Pers\}$ <http://xmlns.com/foaf/0.1/name> $\{Name\}$
$WHERE\ Type =$ <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Document>

ARQ encoding of query OQ_3

$SELECT\ DISTINCT\ ?Name\ ?Doc$
$WHERE$ {
  $?Doc$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://sw.deri.org/~aharth/2004/07/dblp/dblp.owl#Document>.
  $?Doc$ <http://purl.org/dc/elements/1.1/creator> $?Pers.$
  $?Pers$ <http://xmlns.com/foaf/0.1/name> $?Name.$
}

A.3.4 Encodings for query OQ_4

The encodings of query OQ_4 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_4

```
result4(Name, N) :-
    triple(Pers, "http://xmlns.com/foaf/0.1/name", Name),
    N = #count{Art : triple(Art, "http://purl.org/dc/elements/1.1/creator", Pers)},
    N > 0.
```

MULGARA encoding of query OQ_4

```
SELECT $Name
    count ( SELECT $Art
            FROM <rmi://localhost/server1#dblp>
            WHERE $Art <http://purl.org/dc/elements/1.1/creator> $Pers )
FROM <rmi://localhost/server1#data5>
WHERE $Pers <http://xmlns.com/foaf/0.1/name> $Name
AND $Art <http://purl.org/dc/elements/1.1/creator> $Pers;
```

A.3.5 Encodings for query OQ_5

The encodings of query OQ_5 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_5

```
#maxint = 20.

dist(Aut1, Aut2, 1) :-
    triple(Res, "http://purl.org/dc/elements/1.1/creator", Aut1),
    triple(Res, "http://purl.org/dc/elements/1.1/creator", Aut2),
    Aut1! = Aut2.

dist(Aut1, Aut2, Dist) :- dist(Aut1, Aut, Dist1), dist(Aut, Aut2, 1),
    Aut1! = Aut2, Aut1! = Aut, Aut! = Aut2, Dist = Dist1 + 1.

minDist(Aut1, Aut2, Dist) :- dist(Aut1, Aut2, Dist), not haveLessDist(Aut1, Aut2, Dist).

haveLessDist(Aut1, Aut2, Dist1) :- dist(Aut1, Aut2, Dist1), dist(Aut1, Aut2, Dist2), Dist1 > Dist2.

minDistName(Name1, Name2, Dist) :- minDist(Aut1, Aut2, Dist),
    triple(Aut1, "http://xmlns.com/foaf/0.1/name", Name1),
    triple(Aut2, "http://xmlns.com/foaf/0.1/name", Name2).

minDistName("Georg Gottlob", "Paolo Liberatore", Dist)?
```

A.4 Testing on querying of LUBM ontology

In the following we provide the encodings exploited for LUBM queries. In order to better identify the original LUBM queries described at `http://swat.cse.lehigh.edu/projects/lubm/query.htm` we give also their original numbering as LUBM-QueryX.

A.4.1 Encodings for query OQ_6 (LUBM-Query1)

The encodings of query OQ_6 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_6

```
result1(Stud) :-
    triple(Stud, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent"),
    triple(Stud, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse",
            "http://www.Department0.University0.edu/GraduateCourse0").
```

MULGARA encoding of query OQ_6

```
SELECT $Stud
FROM <rmi://localhost/server1#lubm>
WHERE $Stud
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>
AND $Stud
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse>
    <http://www.Department0.University0.edu/GraduateCourse0>;
```

SESAME1 encoding of query OQ_6

```
SELECT DISTINCT Stud
FROM {Stud} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type},
    {Stud} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse> {Cour}
WHERE Type = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>
AND Cour = <http://www.Department0.University0.edu/GraduateCourse0>
```

SESAME2 and ARQ encoding of query OQ_6

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX pref : <http://www.Department0.University0.edu/>

SELECT ?Stud
WHERE {
    ?Stud rdf:type ub:GraduateStudent.
    ?Stud ub:takesCourse pref:GraduateCourse0.
}
```

A.4.2 Encodings for query OQ_7 (LUBM-Query2)

The encodings of query OQ_7 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_7

```
result2(Stud, Uni, Dep) :-
    triple(Stud, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl#GraduateStudent"),
    triple(Uni, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#University"),
    triple(Dep, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Department"),
    triple(Stud, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf", Dep),
    triple(Dep, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#subOrganizationOf", Uni),
    triple(Stud, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#undergraduateDegreeFrom", Uni).
```

MULGARA encoding of query OQ_7

```
SELECT $Stud $Uni $Dep
FROM <rmi://localhost/server1#lubm>
WHERE $Stud
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>
AND $Uni
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#University>
AND $Dep
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Department>
AND $Stud
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf> $Dep
AND $Dep
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#subOrganizationOf> $Uni
AND $Stud
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#undergraduateDegreeFrom> $Uni;
```

SESAME1 encoding of query OQ_7

```
SELECT DISTINCT Stud, Uni, Dep
FROM {Stud} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {TypeS},
    {Uni} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {TypeU},
    {Dep} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {TypeD},
    {Stud} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf> {Dep},
    {Dep} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#subOrganizationOf> {Uni},
    {Stud} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#undergraduateDegreeFrom> {Uni}
WHERE TypeS = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>
AND TypeU = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#University>
AND TypeD = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Department>
```

SESAME2 and ARQ encoding of query OQ_7

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>

SELECT ?Stud ?Uni ?Dep
WHERE {
    ?Stud rdf:type ub:GraduateStudent.
    ?Uni rdf:type ub:University.
    ?Dep rdf:type ub:Department.
    ?Stud ub:memberOf ?Dep.
    ?Dep ub:subOrganizationOf ?Uni.
    ?Stud ub:undergraduateDegreeFrom ?Uni.
}
```

A.4.3 Encodings for query OQ_8 (LUBM-Query3)

The encodings of query OQ_8 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_8

```
result3(Pub) :-
    triple(Pub, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Publication"),
    triple(Pub, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#publicationAuthor",
          "http://www.Department0.University0.edu/AssistantProfessor0").
```

MULGARA encoding of query OQ_8

```
SELECT $Pub
FROM <rmi://localhost/server1#lubm>
WHERE $Pub <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Publication>
AND $Pub <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#publicationAuthor>
    <http://www.Department0.University0.edu/AssistantProfessor0>;
```

SESAME1 encoding of query OQ_8

```
SELECT DISTINCT Pub
FROM {Pub} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type},
    {Pub} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#publicationAuthor> {Aut}
WHERE Type = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Publication>
AND Aut = <http://www.Department0.University0.edu/AssistantProfessor0>
```

SESAME2 and ARQ encoding of query OQ_8

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl#>
PREFIX pref : <http://www.Department0.University0.edu/>

SELECT ?Pub
WHERE {
    ?Pub rdf:type ub:Publication.
    ?Pub ub:publicationAuthor pref:AssistantProfessor0.
}
```

A.4.4 Encodings for query OQ_9 (LUBM-Query4)

The encodings of query OQ_9 exploited in our tests are the following.

DLV^{DB} encoding of query OQ_9

```
result4(Prof, Name, Mail, Tel) :-
    triple(Prof, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”,
            “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Professor”),
    triple(Prof, “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#worksFor”,
            “http://www.Department0.University0.edu”),
    triple(Prof, “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#name”, Name),
    triple(Prof, “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#emailAddress”, Mail),
    triple(Prof, “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#telephone”, Tel).
```

MULGARA encoding of query OQ_9

```
SELECT $Prof $Name $Mail $Tel
FROM <rmi://localhost/server1#lubm>
WHERE $Prof <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Professor>
AND $Prof <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#worksFor>
    <http://www.Department0.University0.edu>
AND $Prof <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#name> $Name
AND $Prof <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#emailAddress> $Mail
AND $Prof <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#telephone> $Tel;
```

SESAME1 encoding of query OQ_9

```
SELECT DISTINCT Prof, Name, Mail, Tel
FROM {Prof} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type},
    {Prof} <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#worksFor> {Uni},
    {Prof} <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#name> {Name},
    {Prof} <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#emailAddress> {Mail},
    {Prof} <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#telephone> {Tel}
WHERE Type = <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Professor>
AND Uni = <http://www.Department0.University0.edu>
```

SESAME2 and ARQ encoding of query OQ_9

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX pref : <http://www.Department0.University0.edu>

SELECT ?Prof ?Name ?Mail ?Tel
WHERE {
    ?Prof rdf:type ub:Professor.
    ?Prof ub:worksFor pref:.
    ?Prof ub:name ?Name.
    ?Prof ub:emailAddress ?Mail.
    ?Prof ub:telephone ?Tel.
}
```

A.4.5 Encodings for query OQ_{10} (LUBM-Query5)

The encodings of query OQ_{10} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{10}

```
result5(Pers) :-
    triple(Pers, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
           "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Person"),
    triple(Pers, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf",
           "http://www.Department0.University0.edu").
```

MULGARA encoding of query OQ_{10}

```
SELECT $Pers
FROM <rmi://localhost/server1#lubm>
WHERE $Pers <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Person>
AND $Pers <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf>
    <http://www.Department0.University0.edu>;
```

SESAME1 encoding of query OQ_{10}

```
SELECT DISTINCT Pers
FROM {Pers} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type},
    {Pers} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf> {Uni}
WHERE Type = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Person>
AND Uni = <http://www.Department0.University0.edu>
```

SESAME2 and ARQ encoding of query OQ_{10}

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX pref : <http://www.Department0.University0.edu>

SELECT ?Pers
WHERE {
    ?Pers rdf:type ub:Person.
    ?Pers ub:memberOf pref:.
}
```

A.4.6 Encodings for query OQ_{11} (LUBM-Query6)

The encodings of query OQ_{11} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{11}

```
result6(Stud) :-
    triple(Stud, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
           "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Student").
```

MULGARA encoding of query OQ_{11}

```
SELECT $Stud
FROM <rmi://localhost/server1#lubm>
WHERE $Stud <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Student>;
```

SESAME1 encoding of query OQ_{11}

```
SELECT DISTINCT Stud
FROM {Stud} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type}
WHERE Type = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Student>
```

SESAME2 and ARQ encoding of query OQ_{11}

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>

SELECT ?Stud
WHERE {
    ?Stud rdf:type ub:Student.
}
```

A.4.7 Encodings for query OQ_{12} (LUBM-Query7)

The encodings of query OQ_{12} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{12}

```
result7(Stud, Cour) :-
    triple(Stud, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”,
          “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Student”),
    triple(Cour, “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”,
          “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Course”),
    triple(“http://www.Department0.University0.edu/AssociateProfessor0”,
          “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#teacherOf”, Cour),
    triple(Stud, “http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#takesCourse”, Cour).
```

MULGARA encoding of query OQ_{12}

```
SELECT $Stud $Cour
FROM <rmi://localhost/server1#lubm>
WHERE $Stud <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
   <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Student>
AND $Cour <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
   <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Course>
AND <http://www.Department0.University0.edu/AssociateProfessor0>
   <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#teacherOf> $Cour
AND $Stud <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#takesCourse> $Cour;
```

SESAME1 encoding of query OQ_{12}

```
SELECT DISTINCT Stud, Cour
FROM {Stud} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {TypeS},
   {Cour} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {TypeC},
   {Prof} <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#teacherOf> {Cour},
   {Stud} <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#takesCourse> {Cour}
WHERE TypeS = <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Student>
AND TypeC = <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Course>
AND Prof = <http://www.Department0.University0.edu/AssociateProfessor0>
```

SESAME2 and ARQ encoding of query OQ_{12}

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#>
PREFIX pref : <http://www.Department0.University0.edu/>

SELECT ?Stud ?Cour
WHERE {
   ?Stud rdf:type ub:Student.
   ?Cour rdf:type ub:Course.
   pref:AssociateProfessor0 ub:teacherOf ?Cour.
   ?Stud ub:takesCourse ?Cour.
}
```

A.4.8 Encodings for query OQ_{13} (LUBM-Query8)

The encodings of query OQ_{13} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{13}

```
result8(Stud, Dep, Mail) :-
    triple(Stud, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Student"),
    triple(Dep, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
          "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Department"),
    triple(Stud, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf", Dep),
    triple(Dep, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#subOrganizationOf",
          "http://www.University0.edu"),
    triple(Stud, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#emailAddress", Mail).
```

MULGARA encoding of query OQ_{13}

```
SELECT $Stud $Dep $Mail
FROM <rmi://localhost/server1#lubm>
WHERE $Stud <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Student>
AND $Stud <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf> $Dep
AND $Dep <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#subOrganizationOf>
    <http://www.University0.edu>
AND $Stud <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#emailAddress> $Mail;
```

SESAME1 encoding of query OQ_{13}

```
SELECT DISTINCT Stud, Dep, Mail
FROM {Stud} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {TypeS},
    {Dep} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {TypeD},
    {Stud} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf> {Dep},
    {Dep} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#subOrganizationOf> {Uni},
    {Stud} <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#emailAddress> {Mail}
WHERE TypeS = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Student>
AND TypeD = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Department>
AND Uni = <http://www.University0.edu>
```

SESAME2 and ARQ encoding of query OQ_{13}

$PREFIX$ *rdf* : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
$PREFIX$ *ub* : <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#>
$PREFIX$ *pref* : <http://www.Department0.University0.edu/>
$PREFIX$ *org* : <http://www.University0.edu>

$SELECT\ ?Stud\ ?Dep\ ?Mail$
$WHERE$ {
  $?Stud$ rdf:type ub:Student.
  $?Dep$ rdf:type ub:Department.
  $?Stud$ ub:memberOf $?Dep$.
  $?Dep$ ub:subOrganizationOf org:.
  $?Stud$ ub:emailAddress $?Mail$.
}

A.4.9 Encodings for query OQ_{14} (LUBM-Query9)

The encodings of query OQ_{14} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{14}

result9($Stud, Adv, Cour$) :-
  triple($Stud$, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
    "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Student"),
  triple(Adv, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
    "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Faculty"),
  triple($Cour$, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
    "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Course"),
  triple($Stud$, "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#advisor", Adv),
  triple($Stud$, "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#takesCourse", $Cour$),
  triple(Adv, "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#teacherOf", $Cour$).

MULGARA encoding of query OQ_{14}

$SELECT\ \$Stud\ \$Adv\ \$Cour$
$FROM$ <rmi://localhost/server1#lubm>
$WHERE\ \$Stud$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Student>
$AND\ \$Adv$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Faculty>
$AND\ \$Cour$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#Course>
$AND\ \$Stud$ <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#takesCourse> $\$Cour$
$AND\ \$Adv$ <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#teacherOf> $\$Cour;$

SESAME1 encoding of query OQ_{14}

$SELECT\ DISTINCT\ Stud,\ Adv,\ Cour$
$FROM\ \{Stud\}$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> $\{TypeS\}$,
  $\{Adv\}$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> $\{TypeF\}$,
  $\{Cour\}$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> $\{TypeC\}$,
  $\{Stud\}$ <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#advisor> $\{Adv\}$,
  $\{Stud\}$ <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse> $\{Cour\}$,
  $\{Adv\}$ <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#teacherOf> $\{Cour\}$
$WHERE\ TypeS =$ <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Student>
$AND\ typeF =$ <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Faculty>
$AND\ typeC =$ <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#Course>

SESAME2 and ARQ encoding of query OQ_{14}

$PREFIX\ rdf:$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
$PREFIX\ ub:$ <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
$PREFIX\ pref:$ <http://www.Department0.University0.edu/>

$SELECT\ ?Stud\ ?Adv\ ?Cour$
$WHERE\ \{$
  $?Stud$ rdf:type ub:Student.
  $?Adv$ rdf:type ub:Faculty.
  $?Cour$ rdf:type ub:Course.
  $?Stud$ ub:advisor $?Adv$.
  $?Stud$ ub:takesCourse $?Cour$.
  $?Adv$ ub:teacherOf $?Cour$.
$\}$

A.4.10 Encodings for query OQ_{15} (LUBM-Query14)

The encodings of query OQ_{15} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{15}

$result14(Stud)$:-
  $triple(Stud,$ “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”,
    “http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#UndergraduateStudent”$)$.

MULGARA encoding of query OQ_{15}

$SELECT\ \$Stud$
$FROM$ <rmi://localhost/server1#lubm>
$WHERE\ \$Stud$ <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#UndergraduateStudent>;

SESAME1 encoding of query OQ_{15}

```
SELECT DISTINCT Stud
FROM {Stud} <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> {Type}
WHERE Type = <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#UndergraduateStudent>
```

SESAME2 and ARQ encoding of query OQ_{15}

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX pref : <http://www.Department0.University0.edu/>

SELECT ?Stud
WHERE {
    ?Stud rdf:type ub:UndergraduateStudent.
}
```

A.4.11 Encodings for query OQ_{16}

The encodings of query OQ_{16} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{16}

```
result16(Aut, N) :-
    triple(Aut, "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#name", Name),
    N = #count{Pub : triple(Pub,
          "http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#publicationAuthor", Aut)},
    N > 0.
```

MULGARA encoding of query OQ_{16}

```
SELECT $Aut
    count ( SELECT $Pub
            FROM <rmi://localhost/server1#lubm>
            WHERE $Pub
                <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#publicationAuthor> $Aut )
FROM <rmi://localhost/server1#lubm5>
WHERE $Aut <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#name> $Name
AND $Pub <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#publicationAuthor> $Aut;
```

ARQ encoding of query OQ_{16}

```
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub : <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>

SELECT ?Aut count(*)
WHERE {
    ?Pub ub:publicationAuthor ?Aut.
    ?Aut ub:name ?Name.
}
GROUP BY ?Aut
```

A.4.12 Encodings for query OQ_{17}

The encodings of query OQ_{17} exploited in our tests are the following.

DLV^{DB} encoding of query OQ_{17}

$\#maxint = 20.$

$dist(Aut1, Aut2, 1)$:-
  $triple(Res,$ "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#publicationAuthor", $Aut1),$
  $triple(Res,$ "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#publicationAuthor", $Aut2),$
  $Aut1! = Aut2.$

$dist(Aut1, Aut2, Dist)$:- $dist(Aut1, Aut, Dist1), dist(Aut, Aut2, 1),$
  $Aut1! = Aut2, Aut1! = Aut, Aut! = Aut2, Dist = Dist1 + 1.$

$minDist(Aut1, Aut2, Dist)$:- $dist(Aut1, Aut2, Dist), not\ haveLessDist(Aut1, Aut2, Dist).$

$haveLessDist(Aut1, Aut2, Dist1)$:- $dist(Aut1, Aut2, Aut1), dist(Aut1, Aut2, Dist2), Dist1 > Dist2.$

$minDistName(Name1, Name2, Dist)$:- $distmin(Aut1, Aut2, Dist),$
  $triple(Aut1,$ "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#name", $Name1),$
  $triple(Aut2,$ "http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#name", $Name2).$

$distminname($"FullProfessor2", "FullProfessor3", $Dist)?$

A.5 Testing on a combinatorial problem

A.5.1 Encodings for query $FastFoods$

The encodings of query $FastFoods$ exploited in our tests are the following.

DLV^{DB} and DLV encoding of query $FastFoods$

```
prestaurant(K) :- restaurant(R,K).
distance(K1,K2,K3) :- prestaurant(K1), prestaurant(K2), K1 > K2, K1 = K2 + K3.
distance(K1,K2,K3) :- prestaurant(K1), prestaurant(K2), K1 <= K2, K2 = K1 + K3.

p2depot(K) :- depot(R,K).
aggr1(RK,Y) :- p2depot(DK1), distance(DK1,RK,Y).
serves(Rname,Dist) :- restaurant(Rname,RK), p2depot(DK), distance(RK,DK,Dist),
          Dist = #min {Y : aggr1(RK,Y) }.

p1depot(R) :- depot(R,K).
altdepot(R,K) ∨ naltdepot(R,K) :- restaurant(R,K), not p1depot(R).

ndepots(N) :- #count {D,K: depot(D,K)} = N, #int(N).
```

```
q(1) :- ndepots(N), #count {D,K: altdepot(D,K)} = N.
 :- not q(1).

paltdepot(K) :- altdepot(R,K).
aggr2(RK,Y) :- paltdepot(DK1), distance(DK1,RK,Y).
altserves(Rname,Dist) :- restaurant(Rname,RK), paltdepot(DK), distance(RK,DK,Dist),
            Dist = #min {Y : aggr2(RK,Y) }.

cost(C) :- #sum {Dist,R : serves(R,Dist)} = C, #int(C).

p(1) :- cost(Cost), #sum {Dist,R : altserves(R,Dist)} <= Cost.
 :- not p(1).
```

A.6 Testing on data transformation problems

A.6.1 Encodings for query $Int2Bin$

The encodings of query $Int2Bin$ exploited in our tests are as follows. For simplicity, we show here only the case for 5 bits. The other encodings have the same structure.

DLV^{DB} encoding of query $Int2Bin$ without functions

```
#maxint=31.
digit(0).
digit(1).

binary(GR,DT4,DT3,DT2,DT1,DT0) :- digit(DT4), digit(DT3), digit(DT2), digit(DT1), digit(DT0),
            N = DT4*16, O = DT3*8, P = DT2*4, Q = DT1*2, R = DT0*1, V = N+PART3,
            PART3 = PART2+PART1, PART2 = O+P, PART1 = Q+R.

dati_esamiB(RN,CI,EC,DT,DT4,DT3,DT2,DT1,DT0,RE,AY) :-
            dati_esamiD(RN,CI,EC,DT,GR,RE,AY), binary(GR,DT4,DT3,DT2,DT1,DT0).
```

DLV^{DB} encoding of query $Int2Bin$ with functions

```
dati_esamiB(RN,CI,EC,DT,BI,RE,AY) :-
            dati_esamiD(RN,CI,EC,DT,GR,RE,AY), #dbo.IntToBin(GR, 5, BI).
```

Next we present the code of external function $IntToBin$ implemented on the *SqlServer* database set as working DB for DLV^{DB}.

```
SET ANSI_NULLS ON
SET QUOTED_IDENTIFIER ON
GO
CREATE FUNCTION [dbo].[IntToBin]
(
        @dec INT,
        @nbit INT
)
RETURNS VARCHAR (20)
AS
BEGIN
        DECLARE @result VARCHAR (20)
        DECLARE @tmp VARCHAR (1)
        DECLARE @quotient INT
        DECLARE @base INT
        DECLARE @remanider INT
        DECLARE @count INT
        SET @quotient=@dec;
        SET @base=2;
        SET @remanider=0;
        SET @count=0;
        SET @result='';
        WHILE @quotient <> 0 AND @count < @nbit
        BEGIN
             SET @remanider=@quotient%@base
             SET @quotient=@quotient/@base
             SET @tmp=CAST(@remanider AS VARCHAR(1))
             SET @result=@tmp+@result
             SET @count=@count+1
        END;
        IF @count < @nbit
        BEGIN
             WHILE @nbit-@count <> 0
             BEGIN
                  SET @result='0'+@result
                  SET @count=@count+1
             END;
        END;
        RETURN @result;
END;
```

A.7 Testing on string similarity computation

A.7.1 Encodings for query $HammingDistances$

The encodings of query $HammingDistances$ exploited in our tests are as follows.

DLV^{DB} encoding of query $HammingDistances$ without functions

Note that, in this encoding, a string is represented by a set of $string(ID, CHAR, POS)$ predicates.

```
hd(ID1,ID2,H) :- string(ID1,P1,C1), string(ID2,P2,C2), ID1 < ID2,
        #count{POS : string(ID1,CHAR1,POS), string(ID2,CHAR2,POS),
                                        CHAR1 ! = CHAR2} = H.
```

DLV^{DB} encoding of query $HammingDistances$ with functions

Note that, in this encoding, a string is represented by a set of $string(ID, S)$ predicates.

```
hd(ID1,ID2,H) :- string(ID1,S1), string(ID2,S2), ID1 < ID2,
                 #dbo.hamming(S1,S2,H).
```

Next we present the code of external function $hamming$ implemented on the *SqlServer* database set as working DB for DLV^{DB}.

```
SET ANSI_NULLS ON
SET QUOTED_IDENTIFIER ON
GO
CREATE FUNCTION [dbo].[hamming]
(
        @s1 VARCHAR (250),
        @s2 VARCHAR (250)
)
RETURNS INT
AS
BEGIN
        DECLARE @count INT,
        DECLARE @index INT,
        DECLARE @len INT,
        DECLARE @s1_sub VARCHAR (250),
        DECLARE @s2_sub VARCHAR (250)
        IF DATALENGTH(@s1) = DATALENGTH(@s2)
        BEGIN
             SET @len=DATALENGTH(@s1)
             SET @index=1
             SET @count=0
             WHILE @index <= @len
             BEGIN
                  SET @s1_sub=SUBSTRING(@s1,@index, @index)
                  SET @s2_sub=SUBSTRING(@s2,@index, @index)
                  SET @index=@index+1
                  IF ASCII(@s1_sub) != ASCII(@s2_sub)
                  BEGIN
                       SET @count=@count+1
                  END
             END
        END
        ELSE
        BEGIN
             SET @count=-1
        END
        RETURN @count
END
```

Bibliography

General References

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

[2] T. Adams, G. Noble, P. Gearon, and D. Wood. MULGARA homepage. `http://www.mulgara.org/`, since 2006.

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

[4] C. Anger, K. Konczak, and T. Linke. `NoMoRe`: A system for non-monotonic reasoning under answer set semantics. In W. F. 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.

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

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

[7] 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.

[8] M. Arenas, L. Bertossi, and J. Chomicki. Specifying and Querying Database Repairs using Logic Programs with Exceptions. In *Proc. of the Fourth International Conference on Flexible Query Answering Systems (FQAS 2000)*, pages 27–41, 2000.

[9] ARQ homepage. `http://jena.sourceforge.net/ARQ/`, since 2004.

[10] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC*, pages 722–735, 2007.

[11] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[12] 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.

[13] 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.

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

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

[16] C. Beeri and R. Ramakrisnhan. On the power of magic. *Journal of Logic Programming*, 10(1-4):255–259, 1991.

[17] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[18] R. Bihlmeyer, W. Faber, C. Koch, N. Leone, C. Mateis, and G. Pfeifer. `dlv` – an overview. In U. Egly and H. Tompits, editors, *Proceedings of the 13th Workshop on Logic Programming (WLP'98)*, pages 65–67, Vienna, Austria, October 1998.

[19] C. Bizer. D2r map - a database to RDF mapping language. in *www (posters)*, 2003.

[20] A. Calì, D. Calvanese, G. D. 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.

[21] A. Calì, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *Int. Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 16–21, 2003.

[22] F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50:333–361, 2007.

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

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

[25] 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.

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

[27] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. 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.

[28] D2r server publishing the DBLP bibliography database.
`http://www4.wiwiss.fu-berlin.de/dblp/`.

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

[30] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System Description: `DLV`. In T. Eiter, W. Faber, and M. aw 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.

[31] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proc. of the 18th Int. Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, Acapulco, Mexico, 2003.

[32] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in DLV. In M. de Vos and A. Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 274–288, Messina, Italy, Sept. 2003. Online at `http://CEUR-WS.org/Vol-78/`.

[33] 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.

[34] A. S. E. Prud'hommeaux. Sparql query language for rdf. w3c candidate recommendation, 14 june 2007. `http://www.w3.org/tr/rdf-sparql-query/`.

[35] 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.

[36] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[37] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. Technical Report INFSYS RR-1843-01-12, Institut für Informationssysteme, Technische Universität Wien, Dec. 2001.

[38] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. Technical Report INFSYS RR-1843-01-11, Institut für Informationssysteme, Technische Universität Wien, Dec. 2001.

[39] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[40] N. L. et al. The infomix system for advanced integration of incomplete and inconsistent data. In *Proc. of 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, 2005. ACM Press.

[41] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In I. O. 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.

[42] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In M. Gelfond, N. Leone, and G. 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.

[43] W. Faber, N. Leone, and G. 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, Aug. 2001. Morgan Kaufmann Publishers.

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

[45] W. Faber and G. Pfeifer. `DLV` homepage, since 1996. `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[46] D. Fensel, W. Wahlster, H. Lieberman, and J. Hendler, editors. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2002.

[47] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *TPLP*, 5(1-2):45–74, 2005.

[48] E. Franconi, A. L. Palma, N. Leone, S. Perri, and F. Scarcello. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In *In* Proc. of LPAR 2001, pages 561–578, 2001.

[49] 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.

[50] 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.

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

[52] H. Gallaire, J. Minker, and J. Nicolas. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16(2):153–186, 1984.

[53] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[54] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *Clasp* : A conflict-driven answer set solver. In *Int. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Tempe, AZ, USA*, pages 260–265, 2007.

[55] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In *LPNMR*, pages 136–148, 2007.

[56] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Jan. 2007. 386–392.

[57] M. Gebser, T. Schaub, and S. Thiele. GrinGo : A new grounder for answer set programming. In *Int. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Tempe, AZ, USA*, pages 266–271, 2007.

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

[59] A. V. 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.

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

[61] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Jornal of Automated Reasoning*, 36(4):345–377, 2006.

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

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

[64] S. Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE Transaction on Knowledge and Data Engineering*, 15(2):368–385, 2003.

[65] 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.

[66] 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.

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

[68] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. *TOCL*, 7(1):1–37, 2006.

[69] T. Janhunen, I. Niemela, P. Simons, and J.-H. You. Partiality and disjunctions in stable model semantics. In A. G. Cohn, F. Giunchiglia, and B. 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.

[70] 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.

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

[72] 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.

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

[74] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *ICLP*, pages 451–465, 2003.

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

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

[77] 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.

[78] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS 2002*, pages 233–246, 2002.

[79] N. Leone, G. Greco, G. Ianni, V. Lio, G. Terracina, T. Eiter, W. Faber, M. Fink, G. Gottlob, R. Rosati, D. Lembo, M. Lenzerini, M. Ruzzi, E. Kalka, B. Nowicki, and W. Staniszkis. The infomix system for advanced integration of incomplete and inconsistent data. In *Proc. of 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, 2005. ACM Press.

[80] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, July 2006.

[81] N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, 1997.

[82] M. Ley. Digital bibliography and library project `http://dblp.uni-trier.de/`.

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

[84] F. Lin and Y. Zhao. Assat: Computing answer sets of a logic program by sat solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002. AAAI Press / MIT Press.

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

[86] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[87] T. Linke. Graph theoretical characterization and computation of answer sets. In B. Nebel, editor, *International Joint Conference on Artificial Intelligence*, pages 641–645, 2001.

[88] 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.

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

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

[91] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 289–300, Philadelphia, Pennsylvania, USA, 2005.

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

[93] LUBM homepage. `http://swat.cse.lehigh.edu/projects/lubm/`.

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

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

[96] 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.

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

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

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

[100] I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. 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.

[101] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. In *Proc. of the 8th Int. Workshop on Non-Monotonic Reasoning (NMR'2000)*, Colorado, USA, April 2000.

[102] S. M. noz, J. Pérez, and C. Gutiérrez. Minimal deductive systems for RDF. in *eswc*, pages 53–67, 2007.

[103] L. Palopoli, S. Rombo, and G. Terracina. Flexible pattern discovery with (extended) disjunctive logic programming. In *Proc. of 15th International Symposium on Methodologies for Intelligent Systems (ISMIS 2005)*, pages 504–513, Saratoga Springs, New York, USA, 2005. Lecture Notes in Artificial Intelligence (3488), Springer-Verlag.

[104] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. Owl web ontology language semantics and abstract syntax. w3c recommendation, 10 february 2004. `http://www.w3.org/tr/owl-semantics/`.

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

[106] A. Polleres. From sparql to rules (and back). In *In Proceedings of the 16th World Wide Web Conference (WWW2007), Banff, Canada*, 2007. Extended technical report version available at `http://www.polleres.net/publications/GIA-TR-2006-11-28.pdf`.

[107] A. Polleres. Personal communication, 2007.

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

[109] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.

[110] S. P. Radziszowski. Small Ramsey Numbers. *The Electronic Journal of Combinatorics*, 1, 1994. Revision 9: July 15, 2002.

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

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

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

[114] P. Rao, K. Sagonas, T. Swift, D. 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.

[115] RDF resource guide. `http://planetrdf.com/guide/`.

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

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

[118] 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.

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

[120] 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.

[121] D. Seipel and H. Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In A. 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.

[122] SESAME homepage. `http://www.openrdf.org/`, since 2002.

[123] P. Simons. Extending the Stable Model Semantics with More Expressive Rules. In M. Gelfond, N. Leone, and G. 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.

[124] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *AI*, 138:181–234, June 2002.

[125] Sparql implementations. `http://esw.w3.org/topic/sparqlimplementations`.

[126] American National Standards Institute: ANSI/ISO/IEC 9075-1999 (SQL: 1999, Parts 1-5), New York, NY, 1999.

[127] H. Stuckenschmidt and J. Broekstra. Time - space trade-offs in scaling up rdf schema reasoning. in *wise workshops*, pages 172–181, 2005.

[128] T. Syrjänen. Lparse 1.0 user's manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[129] 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.

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

[131] 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.

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

[133] 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.

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

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

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

[137] 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.

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

[139] W3C. Rdf semantics. w3c recommendation 10 february 2004, 2006. `http://www.w3.org/TR/rdf-mt/`.

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

[141] M. Winslett. Raghu Ramakrishnan Speaks Out. *SIGMOD Record*, 35(2):77–85, 2006.

Papers Published by the Author of this Thesis

[142] G. Ianni, A. Martello, C. Panetta, and G. Terracina. Faithful and effective querying of RDF ontologies using DLV^{DB}. In *Proc. of the 4th International Workshop on Answer Set Programming (ASP'07)*, Porto, Portugal, 2007.

[143] G. Ianni, A. Martello, C. Panetta, and G. Terracina. Some experiments on the usage of a deductive database for RDFS querying and reasoning. In *Proc. of the 4th Workshop on Semantic Web Applications and Perspectives (SWAP 2007)*, Bari, Italy, 2007.

[144] G. Ianni, A. Martello, C. Panetta, and G. Terracina. Efficiently querying RDF(S) ontologies with Answer Set Programming. *Journal of Logic and Computation (Special issue).*, To appear.

[145] G. Ianni, C. Panetta, and F. Ricca. Specification of assessment-test criteria through ASP specifications. In *Proc. of Answer Set Programming: Advances in Theory and Implementation (ASP'05)*, Bath, UK, 2005.

[146] N. Leone, W. Faber, A. Bria, F. Calimeri, G. Catalano, S. Cozza, T. Dell'Armi, G. Greco, G. Ianni, G. Ielpa, M. Maratea, C. Panetta, S. Perri, F. Ricca, F. Scarcello, G. Terracina, G. Pfeifer, T. Eiter, and G. Gottlob. DLV: An Advanced System for Knowledge Representation and Reasoning. In *ALP Newsletter*, volume 20, n. 3-4. Editor: E. Pontelli, Area Editor: R. Bagnara, 2007.

[147] N. Leone, V. Lio, C. Panetta, and G. Terracina. DLV^{DB}: a system for the efficient evaluation of datalog programs directly on databases. *Intelligenza Artificiale*, 2006. To appear.

[148] G. Terracina, E. D. Francesco, C. Panetta, and N. Leone. Enhancing a DLP System for Advanced Database Applications. In *Proc. of International Conference on Web Reasoning and Rule Systems (RR 2008)*, pages 119–134, Karlsruhe, Germany, 2008. Lecture Notes in Computer Science, Springer.

[149] G. Terracina, E. D. Francesco, C. Panetta, and N. Leone. Experiencing ASP with real world applications. In *Inproc. of 15th Workshop on Knowledge Representation and Automated Reasoning (RCRA 2008)*, Udine, Italy, 2008.

[150] G. Terracina, N. Leone, V. Lio, and C. Panetta. Adding Efficient Data Management to Logic Programming Systems. In *Proc. of 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 2006)*, pages 524–533, Bari, Italy, 2006. Lecture Notes in Artificial Intelligence (4203), Springer.

[151] G. Terracina, N. Leone, V. Lio, and C. Panetta. Comparing Logic Programming Systems with DBMSs on Recursive Queries. In *Inproc. of 13th Workshop on Knowledge Representation and Automated Reasoning (RCRA 2006)*, Udine, Italy, 2008.

[152] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming (TPLP)*, 8(2):129–165, 2008.