Università degli Studi di Roma "La Sapienza"
Dottorato di Ricerca in Ingegneria Informatica

Efficient Data Integration under Integrity Constraints: a Practical Approach

Ph.D. Thesis

Marco Ruzzi

Author's contact information:

Marco Ruzzi
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
via Salaria, 113
I-00198 Roma, Italy
e-mail: ruzzi@dis.uniroma1.it

Contents

1 Introduction **1**

1.1 The data integration problem 2
1.2 Commercial tools for data integration 4
1.3 Dealing with integrity constraints 4
1.4 Contribution of the thesis 5
1.5 Organization of the thesis 6

2 Formal Framework for Data Integration **7**

2.1 Theoretical Background 7
2.1.1 The Relational Model 7
2.1.2 The Datalog language 8
2.1.3 First Order Logic 10
2.1.4 Complexity classes 11
2.2 A Framework for Semantic Data Integration 12
2.2.1 Syntax 12
2.2.2 Semantics 14

3 State of the Art and Current Trends in Data Integration **17**

3.1 Semantic Data Integration 17
3.1.1 GAV Data Integration Systems 18
3.1.2 Query Processing in LAV 21
3.2 New Architectures For Data Integration 23
3.2.1 Peer-to-peer Data Management Systems (PDMS) 23
3.2.2 Peer-to-peer Data Integration 25
3.2.3 Other approaches to Data Integration 25
3.3 Efficient Query Answering Under Integrity Constraints 26
3.3.1 First-order Query Rewriting for Inconsistent Database 26
3.3.2 Querying Inconsistent Database through Probabilistic approaches 27
3.3.3 Other approaches to Efficient Query Answering 28

4 Commercial Systems for Data Integration **31**

4.1 Oracle 10g Information Integration 32

4.2 Microsoft SQL Server 2005 Integration Service . . . 35
4.3 IBM DB2 Information Integrator . . . 36
4.4 Discussion . . . 39

5 An Efficient System for Data Integration **41**
5.1 Limits of Commercial Systems w.r.t. Semantic Data Integration . . . 41
5.2 Architecture of the system . . . 42
5.2.1 System Specification . . . 43
5.2.2 Specification Processor . . . 44
5.2.3 Sources Processor . . . 45
5.2.4 Mapping Processor . . . 47
5.2.5 Query Reformulator . . . 47
5.3 Data integration techniques . . . 47
5.3.1 Internal technique: implementing data integration using views . . . 48
5.3.2 External technique: external management of global schema and mapping 51
5.3.3 Correctness . . . 52

6 Efficient Query Answering for Data Integration Systems **53**
6.1 Limits of standard semantics for Data Integration Systems . . . 54
6.2 Consistent Query Answering . . . 55
6.3 Efficient Algorithms for CQA . . . 56
6.3.1 Formal framework for CQA . . . 57
6.3.2 First-order reducibility of CQs under KDs . . . 60
6.3.3 Computing consistent answers of U-acyclic UCQs . . . 70
6.3.4 Computing consistent answers for weakly-U-cyclic UCQs . . . 76
6.3.5 First-order reducibility of CQs under KDs and IDs . . . 80
6.3.6 First-order reducibility of CQs under KDs and EDs . . . 83
6.4 Efficient Data Integration under Integrity Constraints . . . 89

7 Experiments **93**
7.1 Experiment Scenario . . . 93
7.2 Comparing techniques for handling mapping . . . 95
7.3 Experimenting the efficiency of FOL rewritings for CQA . . . 98
7.4 Testing source access . . . 100

8 Conclusions **103**

List of Figures

4.1 The Oracle 10g Information Integrator Architecture 32
4.2 SQL Server Integration Service Architecture 36
4.3 The DB2II Architecture 37
4.4 Heterogeneous Data Source Support 39
4.5 Systems Comparison 40

5.1 System architecture 42
5.2 The behavior of the system 44
5.3 Example Data Integration System Specification 45
5.4 View definition for the player global relation of Example 5.2.3 47
5.5 View definition for the team global relation of Example 5.2.3 48
5.6 $compile_i(\mathcal{I})$ algorithm. 49
5.7 The $translate_i(Q, \mathcal{I})$ algorithm. 50
5.8 $translate_e(q, \mathcal{I})$ algorithm 52

6.1 Join Graphs for queries of Example 6.3.5 62
6.2 Typed join graph for query q of Example 6.3.11 64
6.3 The algorithm NodeRewrite 65
6.4 Typed join graph for query $q :- r(X, Y), s(Y, X)$. 66
6.5 The computeConsAnswers algorithm 68
6.6 The goodImages algorithm 69
6.7 Typed join graph for the disjuncts of Q of Example 6.3.23 72
6.8 The algorithm NodeRewriteNew 77
6.9 The ID-rewrite algorithm 81
6.10 The FolKIDRewrite algorithm 82
6.11 The algorithm FolTree 88
6.12 Typed join graph for the query of Example 6.3.2 89
6.13 The new architecture of the system 91

7.1 Relational schema of the test database 94
7.2 Tables size (number of tuples) 95
7.3 GAV mappings for the test data integration system 96
7.4 Experimental Results 97

7.5 Experimental Results 99
7.6 Experimental Results 101
7.7 Experimental Results 102

Chapter 1

Introduction

Data are the most relevant part of an information system. The need to handle huge amounts of data leads software companies towards the development of tools that allow several types of organization to effectively face the *on demand* challenge [2, 56]: data changes and this happens continuously and faster than ever before. Moreover, data come from an increasing number and variety of information sources, and the data management infrastructure of a generic business entity should be able to share its data with other entities, integrating them with information coming from elsewhere, also surmounting various kinds of heterogeneity.

Several efforts are made in this direction from both the software development [57, 3] and the academic field [70, 92]. In particular, research in *information integration* aims at providing robust and effective solutions to the problem of accessing data not necessarily locally managed by the system with which the user interact. Unfortunately, "information integration" often assumes very different meanings. Software industry mostly faces the matter from a procedural point of view: *data federation* tools [15], for example, adopt a database management system as a kind of middleware infrastructure that uses a set of software modules (wrappers) to access heterogeneous data sources. Basically, data federation tools provide the user with a single database obtained by simply putting together the databases provided by each data source. As another example, a *grid computing* tool [27] provides a framework that enables data owners to cope with the localization of data in distributed environments. On the other hand, academic research focuses on theoretical solutions, concerning topics like data modelling, expressiveness of formalisms, semantic characterization of the problem, which constitute very difficult tasks.

The present work is somehow located in the middle between academic and industry approaches: more specifically, our aim is to find a way to use solutions coming from both the aforementioned approaches, providing an efficient and effective solution to the information integration problem. Therefore, the main goal of our work is to build a novel and efficient system that is able to really access heterogeneous data sources, masking them under a common and expressive representation, and enabling users to efficiently retrieve useful answers to their queries. To this aim, we have analyzed three commercial tools for data integration, Oracle 10g Information Integration, Microsoft SQL Server 2005 and IBM DB2 Information Integrator, in order to devise a way to access different data sources as if they were a single resource, regardless of

where the information resides, while retaining the autonomy and integrity of the sources' content. Furthermore, we want to mask the integrated data under a mediated schema enriched with expressive forms of integrity constraints, which allow for suitably modelling a variety of real world situations.

Our goal is very ambitious: in fact, although some research works are in progress [47, 37, 71, 46], there are still many problems to solve. On one hand, commercial tools are rather far from meeting the requirements coming from the semantic data integration theory (especially for the lack of support in specifying a mediated schema and processing user queries issued over it); on the other hand, solutions provided for that problems are still inefficient [19, 20].

In the rest of this chapter we will sketch the basic issues studied in this thesis. In particular we will introduce the data integration problem in Section 1.1, and provide an overview of the commercial solutions available for such problem in Section 1.2. In Section 1.3 we introduce the issue of dealing with integrity constraint in data integration systems. Finally, in Section 1.4 we present the contributions of this thesis, and describe its organization in the last section of this chapter.

1.1 The data integration problem

Information integration has emerged as a crucial issue in many application domains, e.g., distributed databases, cooperative information systems, data warehousing, data mining, data exchange, as well as in accessing distributed data over the web.

In particular, data integration is the problem of combining data residing at different sources, and providing the user with a unified view of these data [60, 62, 93, 70]. The *global (or mediated) schema* constitutes such unified view, representing the intensional level of the integrated data, and is the schema over which the users express their queries. A data integration system, then, provides users with a global schema and frees them, when they formulate their queries, from the need of knowing where data are really stored, how data are structured at the sources, and how data are to be merged and reconciled to fit into the global schema.

Sources that have to be integrated in a data integration system are typically heterogeneous, that is, they adopt different models and systems for storing data. This poses challenging problems in both representing the sources in a common format within the integration system, and specifying the global schema. As for the former issue, data integration systems make use of suitable software components, called *wrappers*, that present data at the sources in the form adopted within the system, hiding the original structure of the sources and the way in which they are modelled. The representation of the sources in the system, is generally given in terms of a *source schema*.

In order to create a correspondence between real data stored at the sources, and global data accessible by the user through the global schema, *mappings* must be defined between global and source elements. One of the most adopted formalisms in representing such mappings, consists in associating a query over the global schema with a query over the source schema [70]. Depending on the various forms of such queries, the mappings assume well-known structures: in the *global-as-view* (GAV) paradigm, a single element of the global schema is associated with a query over the elements of the source schema. Conversely, according to the *local-as-view* (LAV) paradigm, a

single source element is associated with a query over the global schema. Finally, in the *generalized-local-as-view* (GLAV) paradigm queries over the source and global schema can be arbitrary.

The way in which mapping assertions are interpreted assumes particular importance in defining the semantics of the data integration system. According to the literature, three different assumptions can be made on the mapping assertions: under the *sound* assumption, data provided by the sources are interpreted as a subset of the global data. Vice versa, the mapping is assumed *complete*, when sources data provide a superset of the data of the global schema. The mapping is assumed *exact*, when it is both sound and complete.

Another important aspect in a data integration system is whether the system is able to materialize data retrieved from the sources (through the mappings). In the *materialized* approach, the system computes the extension of the structures in the global schema by replicating the data at the sources. Obviously, maintenance of replicated data against updates at the sources is a central aspect in this context. A possible way to deal with this problem is to recompute materialized data when the sources change, but it could be extremely expensive and impractical for dynamic scenarios. In this work we restrict our attention to *virtual* data integration systems, in which data are *not* materialized outside the sources in which they are originally stored, and are accessed only during query processing. This approach is indeed extensively under investigation in the last years due to the always more growing request on information on-demand. Furthermore, many results we will discuss in the present works, are relevant also for the materialized approach We refer the reader to [96, 63, 12] for specific literature for materialized data integration.

A final issue to be discussed to complete our introduction to the data integration problem is query answering [59, 60, 70, 92]. Answering user queries is the most important task accomplished by a data integration system: user queries posed over the global schema must be answered only on the basis of the data at the sources and the knowledge provided by the global schema and the mapping. Several approaches have been proposed to deal with this problem.

One of the most adopted techniques aims at reformulating user queries so as they can be expressed directly over the source data. The effective employability of such a technique strongly depends on how the mapping is designed and on which kinds of integrity constraints are specified on the global schema. For GAV data integration systems, in the absence of integrity constraints, this can be accomplished by means of simple *unfolding* techniques: basically this consists in unfolding each global element occurring in a user query with its definition provided by the mapping. For LAV systems, things get harder: indeed, in general, a mapping expressed according to the local-as-view paradigm provides only a partial knowledge about the data satisfying the global schema. In these cases, query reformulation is most difficult than a simple unfolding algorithm, even in the absence of integrity constraints, and it is accomplished by means of complex query rewriting techniques. However the presence of integrity constraints complicates query answering in such a way that also the GAV case can not be resolved by means of unfolding, as we will discuss in Section 1.3.

1.2 Commercial tools for data integration

Many software vendors provide different solutions for the data integration problem, which has been historically faced through database replication systems or data warehouses. However, in the last years, IT systems have been called upon to support data integration in real time so that up-to-date information are always available and ready to use. Some of the central features provided by major software companies concern distributed database systems, efficient technologies for data search and the ability to access a number of heterogeneous data sources.

After a deep analysis of several cutting-edge commercial solutions available on the software market, some limitations emerged that affect the practical applicability of tools for data integration. Indeed, such tools mainly face the integration challenge form a procedural view point: a robust support is provided in order to physically access heterogeneous data sources as if they were standard database tables, but no automatic mechanisms can be found for building a mediated global schema through mappings. Moreover, although some features are provided in order to perform some data transformations while merging various data sources, the possibility to express integrity constraints of any kind over the reconciled data, is still a missing feature. The above cited limitations constitute some of the motivations of the present work.

1.3 Dealing with integrity constraints

The capability of handling integrity constraints is one of the central issues for data integration systems. Indeed, the possibility to express integrity constraints over the global schema yields a more realistic and expressive representation of the users' domain of interest. Moreover, when integrating data coming from multiple independent data sources, we can not expect data to satisfy integrity constraints posed over the global schema, even if original data sources are locally consistent. When this situation arises, the data integration system is said to be *inconsistent*.

An approach to deal with constraint violations amounts to explicitly repair or clean data inconsistencies before performing the integration [13]. However, this is not always possible or convenient, as data sources could be, for example, out of the control of the data integration system. Furthermore, as we already mentioned, data sources could be locally consistent, becoming inconsistent when integrated: in these cases explicit changing of a source could lead to the loss of useful data.

An alternative approach to the explicit repair of data is followed by research in *consistent query answering*. This approach aims at defining the semantics of an inconsistent data integration system in terms of *repairs*, i.e., those global databases that satisfy the integrity constraints expressed on its global schema, and "approximate at best" the semantics of the mapping (i.e., the sound, complete or exact assumptions adopted for its interpretation). Then a *consistent answer* to a query is an answer to a query over each such repair. The main problem of such approaches, is the high computational complexity of computing consistent answers [19, 35]. Therefore, the issue of scalability of *consistent query answering* algorithms is somehow central in the research in this field.

1.4 Contribution of the thesis

The contributions of this thesis can be summarized as follows:

(i) we provide an analysis of the commercial technologies available in the field of data integration. To this aim we break down three comprehensive solutions for the data integration issue, coming from leading software producers: Oracle, IBM and Microsoft. It is well known that such companies provide largely adopted solutions for data storage: Oracle DBMS, SQL Server and DB2 Universal Database are three of the most popular database management systems available on the market. In the latest versions of such products, some solutions to the data integration problem are also provided. It will come out that a discrepancy arises between the meanings that the term *data integration* assumes in the commercial environment, and the meanings it assumes in the corresponding computer science research area. We will point out some limitations enforced by such commercial tools for data integration, investigating possible ways to overcome such limitations.

(ii) Starting from the critic of the commercial solutions for data integration we build a novel system for semantic data integration (in the absence of integrity constraints) which is based on the commercial tools analyzed. In detail, our system is able to build a "simple" data integration system instance over the aforementioned commercial systems with GAV mappings and no integrity constraints expressed over the global schema. We point out that this is a significative improvement of the features of the underlying tools: in fact, the possibility of defining a global schema and a mapping over a set of sources is still a missing feature in all the systems we have considered in this thesis. The system we present implements two different data integration techniques: the first, which exploits the capability of managing views of Oracle, DB2 and SQL Server, and the second which makes use of a rewriting module whose task is to reformulate users' queries on the basis of the mapping.

(iii) In order to enable our system to handle integrity constraints, we define a number of query rewriting algorithms for efficient consistent query answering. In particular, all the algorithms we designed, produce a first-order rewriting of the user query, that encodes the integrity constraints together with the query. The use of first-order rewriting yields two important outcomes: (i) first-order queries can be evaluated very efficiently over inconsistent databases and (ii) can be directly expressed in SQL and evaluated exploiting the SQL engines of the tools adopted. Such study of the first-order rewriting of queries lead us to defining some classes of queries for which the consistent query answering problem is tractable (in fact, in LogSpace in data complexity, i.e., the complexity computed with only the database instance as input).

(iv) By using the former results in the consistent query answering field, we extended the capabilities of our system to the treatment of expressive forms of global schemas with integrity constraints. We considered three of the most commonly adopted forms of integrity constraints: key constrains, inclusion dependencies (and hence, foreign key constraints) and exclusion dependencies.

(v) We conducted several experiments on a real large database containing information about students of the University of Rome "La Sapienza". Some experiments compared the two data integration techniques provided by our system. Furthermore, a comparison has been made between the first-order rewriting approach and a the standard way used to solve consistent query answering problems, which exploits Answer Set Programming technique: it came out an effective improvement of the query answering times, also for a relatively low number of inconsistencies. Further experiments, compared the efficiency of Oracle, DB2 and SQL Server in accessing heterogeneous sources.

1.5 Organization of the thesis

The present work is structured as follows:

- In Chapter 2 we recall the basic theoretical notions necessary for the reading of this thesis (Section 2.1), and provide a formal framework for the arguments studied (Section 2.2).
- In Chapter 3 a summary of the main results presented in the literature for the data integration research topics is presented. In particular, Section 3.1 mainly focuses on the general technologies and systems available for the data integration problem, Section 3.2 present some works facing architectural problems for data integration systems, while in Section 3.3 a panoramic of recent works in the field of efficient query answering is provided.
- In Chapter 4 three commercial systems for data integration are dissected and compared. Furthermore, a critic of such systems is presented with the respect to their effective capabilities of providing solutions for the data integration problem.
- In Chapter 5 a new system for the data integration is presented, which exploits commercial tools in order to provide a robust and expressive solution to the issues of the data integration problem.
- In Chapter 6 several algorithm for efficient consistent query answering are presented, in order to enable the system presented in Chapter 5 to deal with expressive forms of integrity constraints.
- In Chapter 7 a series of experimental results are shown; the experiments have been conducted in order to test the effective applicability of our system in real practical cases.

Chapter 2

Formal Framework for Data Integration

2.1 Theoretical Background

The aim of this section is to recall some basic theoretical notions we will use for the presentation of the arguments of the thesis. We illustrate the Relational Model (Section 2.1.1), the basic notions of the Datalog language (Section 2.1.2) and an introduction of the first order logic language (Section 2.1.3), while in Section 2.1.4 we recall some notions on complexity theory. All the topics covered here are intended to be very introductory, and for further details we refer the reader to [5, 83, 48].

2.1.1 The Relational Model

Consider to have an infinite, fixed database domain $\mathcal{U}$ whose elements can be referenced by means of an infinity of constants $c_1, c_2, \ldots, c_n$. Every constant refers to a unique element of $\mathcal{U}$ (unique name assumption), that is, different constants denote different elements of $\mathcal{U}$. A *relational schema* $\mathcal{R}$ is a pair $\langle \mathcal{A}, \Sigma \rangle$ in which:

- $\mathcal{A}$ is a set of *relation schemas* (or *relations*, or *predicates*), each one constituted by its name and the names of its attributes. We often will represent the attributes of a relation r with integers $(1, \ldots, n)$. The number n of attributes of a relation r is said to be the *arity* of the relation: a relation r of arity n is denoted as r/n.
- Σ is a set of integrity constraints, i.e., a set of expressions over the relation schemas in $\mathcal{A}$. An integrity constraint can be viewed as an assertion that is meant to be satisfied by the instances (see below) of the database schema $\mathcal{A}$.

Given a relation schema $r(A_1, \ldots, A_n)$ and a sequence $\mathbf{A} = A_{i_1}, \ldots, A_{i_m}$ of its attribute, the expression $r[\mathbf{A}]$ denote the projection of r over $\mathbf{A}$.

In the following we will consider Σ as composed by three subset of integrity constraints $\mathcal{K}$, $\mathcal{E}$ and $\mathcal{F}$:

- $\mathcal{K}$ is a set of key dependencies (KDs); given a relation r in $\mathcal{A}$ and a sequence $\mathbf{A} = A_1, \ldots, A_n$ of distinct attributes of r, a *key dependency* over r is an expression of the form $key(r) = \mathbf{A}$. We assume that at most one key dependency is specified for each relation.
- $\mathcal{E}$ is a set of exclusion dependencies (EDs); let r_1 and r_2 be two relations in $\mathcal{A}$ and $\mathbf{A} = A_1, \ldots, A_n$ and $\mathbf{B} = B_1, \ldots, B_n$ respectively two sequences of distinct attributes of r_1 and r_2, an *exclusion dependency* between r_1 and r_2 is an expression of the form $r_1[\mathbf{A}] \cap r_2[\mathbf{B}] = \emptyset$.
- $\mathcal{F}$ is a set of inclusion dependencies (IDs); given two relations of $\mathcal{A}$, r_1 and r_2, and let $\mathbf{A} = A_1, \ldots, A_n$ and $\mathbf{B} = B_1, \ldots, B_n$ be respectively two sequences of attributes of r_1 and r_2, an *inclusion dependency* between r_1 and r_2 is an expression of the form $r_1[\mathbf{A}] \subseteq r_2[\mathbf{B}]$.

A *database instance*, or *database* $\mathcal{DB}$ for a relational schema $\mathcal{R}$ is a set of fact $r(t)$ where r is a relation in $\mathcal{A}$ of arity n, and $t \in \mathcal{U}^n$ is an n-tuple of constants of $\mathcal{U}$. Given a database for $\mathcal{R} = \langle \mathcal{A}, \Sigma \rangle$ and a relation $r \in \mathcal{A}$, we denote with $r^{\mathcal{DB}}$ the set $\{t \mid r(t) \in \mathcal{DB}\}$. Furthermore, given a sequences $\mathbf{A} = A_{i_1}, \ldots, A_{i_m}$ of the attributes of a tuple $t \in r^{\mathcal{DB}}$, we denote with $t[\mathbf{A}]$ the projection of t over $\mathbf{A}$. Given a relational schema $\mathcal{R}$, we say that a database instance $\mathcal{DB}$ is consistent with $\mathcal{R}$ if it satisfies all constraints expressed in $\mathcal{R}$. In details:

- $\mathcal{DB}$ satisfies a key dependency $key(r) = \mathbf{A}$ if for each $t_1, t_2 \in r^{\mathcal{DB}}$ with $t_1 \neq t_2$ we have $t_1[\mathbf{A}] \neq t_2[\mathbf{A}]$.
- $\mathcal{DB}$ satisfies an exclusion dependency $r_1[\mathbf{A}] \cap r_2[\mathbf{B}] = \emptyset$ if there do not exist two tuples $t_1 \in {r_1}^{\mathcal{DB}}$ and $t_2 \in {r_2}^{\mathcal{DB}}$ such that $t_1[\mathbf{A}] = t_2[\mathbf{B}]$.
- $\mathcal{DB}$ satisfies an inclusion dependency $r_1[\mathbf{A}] \subseteq r_2[\mathbf{B}]$ if for each tuple $t_1 \in {r_1}^{\mathcal{DB}}$ there exist a tuple $t_2 \in {r_2}^{\mathcal{DB}}$ such that $t_1[\mathbf{A}] = t_2[\mathbf{B}]$.

In the following we will indicate a relational schema with both the expressions $\langle \mathcal{A}, \Sigma \rangle$ and $\langle \mathcal{A}, \mathcal{K}, \mathcal{E}, \mathcal{F} \rangle$. In the absence of some kinds of dependencies, we will omit the corresponding symbols.

2.1.2 The Datalog language

Variables and constants are *terms*. An *atom* of arity n is an expression of the form $r(T_1, \ldots, T_n)$ in which r is a predicate (or relation) of arity n, and T_1, $T_2, \ldots, T_n$ are terms.

A *positive Datalog*, or simply Datalog, rule ρ, is an expression of the form

$$r(\vec{\mathbf{x}}) \leftarrow r_1(\vec{\mathbf{x}}_1), r_2(\vec{\mathbf{x}}_2), \ldots, r_k(\vec{\mathbf{x}}_k). \tag{2.1}$$

where $k \geq 0$ and $r_1(\vec{\mathbf{x_1}}), r_2(\vec{\mathbf{x_2}}), \ldots, r_k(\vec{\mathbf{x_k}}$ are atoms. $r(\vec{\mathbf{x}})$ is said to be the *head* of the rule, denoted by $head(\rho)$, while $r_1(\vec{\mathbf{x_1}}), r_2(\vec{\mathbf{x_2}}), \ldots, r_k(\vec{\mathbf{x_k}})$ is the body, denoted by $body(\rho)$. All terms appearing in the head of the rule must appear also in the body. A positive Datalog rule with $k > 0$ and in which the relational symbol in the head does not appear in the body is called *conjunctive query*. Moreover, variable appearing in $head(\rho)$ are called *distinguished variables*, while term appearing in $body(\rho)$ and not in $head(\rho)$ are called *non-distinguished variables*. A *bound term* is either a non-distinguished variable appearing in more then one atom of $body(\rho)$, a

constant or a distinguished variable. A non-distinguished variable appearing only once in $body(\rho)$ is said to be *unbound*.

The number of terms appearing in the head of a Datalog rule is said to be the *arity* of the rule: a rule of arity 0 is a *boolean* rule. A *fact* is a rule in which $k = m = 0$.

A $Datalog^{\neg}$ rule is an expression of the form

$$r(\vec{\mathbf{x}}) \leftarrow r_1(\vec{\mathbf{x}}_1), \ldots, r_k(\vec{\mathbf{x}}_k), \text{not } r_{k+1}(\vec{\mathbf{x}}_{k+1}), \ldots, \text{not } r_{k+m}(\vec{\mathbf{x}}_{k+m}). \tag{2.2}$$

in which $k \geq 0, m \geq 0$ and there do not exist $1 \leq i \leq k$ and $k+1 \leq j \leq k+m$ such that $r_i = r_j$. The notion of boolean query and fact are the same as for positive Datalog rules.

A $Datalog^{\neg}$ *program* $\mathcal{P}$ is a finite set of Datalog$^{\neg}$ rules. The program is said to be a *positive* Datalog program if all its rules are positive ($m = 0$). A direct labelled graph $G = \langle V, E \rangle$ can be associated with a program $\mathcal{P}$, in which the set of vertex V coincides with the set of predicates appearing in the rules of the program, and the set of edges E is defined as follows: an edge from the vertex r to the vertex s belongs to E if and only if there exists a rule in $\mathcal{P}$ in which r is in the head of that rule, and s is in its body. Edges referring to positive rules are labelled with "p", whereas edges referring to Datalog$^{\neg}$ rules are labelled with "n". If the graph G is acyclic, the program is said *non-recursive*. If every cycle in G involve only "p"-labelled arcs, then the program is said *stratified* Datalog$^{\neg}$.

A *union of conjunctive query* is a non-recursive Datalog program in which all the rules have the same head predicate, and is written in the form:

$$r(\vec{\mathbf{x}}) \leftarrow conj_1(\vec{\mathbf{x}}_1, \vec{\mathbf{y}}_1) \vee \ldots \vee conj_k(\vec{\mathbf{x}}_k, \vec{\mathbf{y}}_k). \tag{2.3}$$

where for each $i \in \{1, \ldots, k\}$, $conj_i$ is a conjunction of atoms.

Predicate symbols appearing in a program $\mathcal{P}$ can be *extensional* (EBD) or *intensional*(IDB) predicates. The former are defined by the facts of a database, the latter are defined by the rules of the program.

A *substitution* σ is a set of pairs of the form $\{X_1 \rightarrow T_1, \ldots, X_n \rightarrow T_n\}$ in which all X_i are distinct variables and all T_i are terms. Given an atom A (or a rule ρ, or a program $\mathcal{P}$) we denote with $\sigma(A)$ (or $\sigma(\rho)$, or $\sigma(\mathcal{P})$ the atom obtained from A (or ρ, or $\mathcal{P}$) by simultaneously replacing each occurrence of X_i with T_i. Terms, atoms, rules or programs are *ground* if no variable symbols occur in it. Let $\mathcal{U}_{\mathcal{P}}$ be the set of all constants occurring in a program $\mathcal{P}$, and ρ a rule in $\mathcal{P}$, we denote with $ground(\rho)$ the set of rules obtained by applying all the possible substitutions from the variables appearing in ρ to the elements of $\mathcal{U}_{\mathcal{P}}$. An *interpretation* I for a program $\mathcal{P}$ is any subset of $B_{\mathcal{P}}$, where $B_{\mathcal{P}}$ is the set of all ground literals constructible from the predicate symbols occurring in $\mathcal{P}$ and the elements of $\mathcal{U}_{\mathcal{P}}$. A positive ground literal g is evaluated to *true* if and only if $g \in I$; we denote with $value_I(g)$ the value of a ground literal g. Consequently, the value of a negative ground literal g is $\neg value_I(g)$. Given a conjunction of literals $C = g_1, g_2, \ldots, g_n$, we denote with $value_I(C) = min(\{value_I(g_i) \mid 1 \leq i \leq n\})$. If $n = 0$ then $value_I(C) = 0$. Given a ground rule ρ and an interpretation I, we say that ρ is *satisfied* by I if $value_I(head(\rho)) \geq value_I(body(\rho))$. A model M of a program $\mathcal{P}$ is an interpretation that satisfies all ground rules of $\mathcal{P}$.

Let $\mathcal{P}$ be a positive Datalog program, the model-theoretic semantics assigns to $\mathcal{P}$ the set $MM(\mathcal{P})$ of its *minimal models*, where a model M for $\mathcal{P}$ is minimal, if no proper subset of M is a model for $\mathcal{P}$.

The stable model semantics applies also to programs with negation. The *reduct* of a ground program $\mathcal{P}$ with respect to a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by

- deleting each rule having a ground literal *not* p, such that p is in X;
- deleting the negative body from the remaining rules.

An interpretation M is a stable model of $\mathcal{P}$ if and only if $M \in MM(\mathcal{P}^M)$. The set of stable models of $\mathcal{P}$ in denoted by $SM(\mathcal{P})$

A stable model of a program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that X is a minimal model of $ground(P)^X$. The set of all stable models for a program $\mathcal{P}$ is denoted by $SM(\mathcal{P})$. For positive $\mathcal{P}$, stable model and minimal model semantics coincide, i.e. $SM(\mathcal{P}) = MM(\mathcal{P})$. It is well-known that for positive programs minimal models and stable models coincide, and that positive (resp. stratified) programs have a unique minimal (resp. stable) model.

In the following, given a $Datalog^{\neg}$ program $\mathcal{P}$, and a set of facts $\mathcal{D}$ that represent the extension of EDB predicates (i.e., facts of a database), we indicate with $\mathcal{P}^{\mathcal{D}}$ the union $\mathcal{P} \cup \mathcal{D}$, and say that $\mathcal{P}^{\mathcal{D}}$ is the evaluation of $\mathcal{P}$ over $\mathcal{D}$.

2.1.3 First Order Logic

The *first-order logic*[1] language $\mathcal{L}$ is build on the following sets of symbols: (i) the propositional connectives ($\neg, \wedge, \vee, \rightarrow$ e $\leftrightarrow$); (ii) the propositional constants $\top$ and $\bot$; (iii) the equality symbol $=$; (iv) an infinite numerable set of variable symbols $(x_1, x_2, \ldots)$; (v) the universal and existential quantifiers (resp. $\forall$, and $\exists$). Furthermore, the following sets of parameters are also used to build first-order logic formulas:

- a finite or numerable set of *predicate* symbols, each with an associated arity n $(P_1, P_2, \ldots, P_l)$.
- a finite or numerable set of *constant* symbols $(c_1, c_2, \ldots, c_m)$.

A *term* is either a constant or a variable. $\top$ and $\bot$ are *atoms*. If $t_1, t_2, \ldots, t_n$ are terms, then the expressions $t_1 = t_2$ and $P(t_1, \ldots, t_2)$, in which P is a predicate symbol of arity n, are atoms. By using terms and atoms we can build the set of well-formed formulas of the first-order logic language. Every atom is a *formula*. If F is a formula then $\neg F$ is a formula. Given two formulas F and G, and a binary connective $\circ$, then $F \circ G$ is a formula. If F is a formula and x is a variable, $\forall x F$ and $\exists x F$ are formulas. Given an atom A we say that a variable x occurs *free* in A if x occurs in A. We say that a variable x occurs free in a formula $\neg F$ if it occurs free in F. x occurs free in $F \circ G$ if it occurs free in F or in G. A variable x occurs free in a formula $\forall z F$ (resp. $\exists z F$) if occurs free in F and $x \neq z$.

[1] We consider here the function-free first-order language

A *structure* for a first-order language $\mathcal{L}$ is a pair $\mathcal{A} = \langle D, I\rangle$, where D is the *domain* of $\mathcal{A}$ and I is a function called *interpretation* that associates every constant symbol c with an element $c^I \in D$, and every predicate P of arity n, with a relation $P^I \subseteq D^n$. Given a structure $\mathcal{A}$ we denote with $\eta : \mathcal{V} \mapsto D$ a function called *assign* from the set of variables $\mathcal{V}$ of $\mathcal{L}$, to the domain of $\mathcal{A}$. The assign function can be extended in order to assign a value to the terms of the language, obtaining the $\overline{\eta} = \langle I, \eta\rangle$ function such that, if x is a variable, then $x^{I,\eta} = x^\eta$ and if c is a constant, then $c^{I,\eta} = c^I$. Given a formula F, a structure $\mathcal{A}$ and an assign function η, we inductively define when a sentence (i.e., a closed formula) is satisfied by $\mathcal{A}$ and η, as follows:

$$\mathcal{A}, \eta \models F$$

(i) $\mathcal{A}, \eta \models \top$ and $\mathcal{A}, \eta \not\models \bot$[2]; (ii) if F is an atom of the form $P(t_1, \ldots, t_n)$ then $\mathcal{A}, \eta \models P(t_1, \ldots, t_n)$ if and only if $\langle {t_1}^{I,\eta}, \ldots, {t_n}^{I,\eta}\rangle \in P^I$; (iii) if F is an atomic formula of the form $t_1 = t_2$ then $\mathcal{A}, \eta \models (t_1 = t_2)$ if and only if ${t_1}^{I,\eta}$ and ${t_2}^{I,\eta}$ are the same element of D; (iv) $\mathcal{A}, \eta \models \neg F$ if and only if $\mathcal{A}, \eta \not\models F$; (v) $\mathcal{A}, \eta \models F \wedge G$ if $\mathcal{A}, \eta \models F$ and $\mathcal{A}, \eta \models G$; (vi) $\mathcal{A}, \eta \models F \vee G$ if $\mathcal{A}, \eta \models F$ or $\mathcal{A}, \eta \models G$; (vii) $\mathcal{A}, \eta \models F \rightarrow G$ if $\mathcal{A}, \eta \models F$ implies $\mathcal{A}, \eta \models G$; (viii) $\mathcal{A}, \eta \models F \leftrightarrow G$ if $\mathcal{A}, \eta \models F$ and $\mathcal{A}, \eta \models G$ or $\mathcal{A}, \eta \not\models F$ and $\mathcal{A}, \eta \not\models G$; (ix) $\mathcal{A}, \eta \models \forall x F$ if for all $d \in D$ the formula $\mathcal{A} \models F(\eta[d/x])$[3]; (x) $\mathcal{A}, \eta \models \exists x F$ if exists a value $d \in D$ such that the formula $\mathcal{A} \models F(\eta[d/x])$.

Given a formula $F \in \mathcal{L}$, a structure $\mathcal{A}$ is a *model* for F (denoted with $\mathcal{A} \models F$ if $\mathcal{A}, \eta \models F$ for every assign function η. In this case, f is said *true* in $\mathcal{A}$. A formula F that is true for every structure of $\mathcal{L}$, is said *valid* and is denoted with $\models F$.

Let us consider now a first-order formula $F(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are free variables. A *first-order query* q of arity n is an expression of the form $q = \{x_1, \ldots, x_n \mid F(x_1, \ldots, x_n)\}$. Given a first-order query q and an interpretation I, we denote with $q^{\mathcal{D}}$ the *evaluation* of q over I , i.e., $q^I = \{t_1, \ldots, t_n \mid I \models F(t_1, \ldots, t_n)\}$, where each t_i for $i = 1, \ldots, n$ is a constant symbol and $F(t_1, \ldots, t_n)$ is the first-order sentence obtained from F by replacing each free variable x_i with t_i. A *boolean first-order query* is a first-order query with arity $n = 0$. The above semantics of FOL queries naturally extends to the case of FOL queries over relational databases. Given a first-order query q and a database instance $\mathcal{D}$, we denote with $q^{\mathcal{D}}$ the *evaluation* of q over $\mathcal{D}$, i.e., $q^{\mathcal{D}} = \{t_1, \ldots, t_n \mid \mathcal{D} \models F(t_1, \ldots, t_n)\}$, where each t_i for $i = 1, \ldots, n$ is a constant symbol and $F(t_1, \ldots, t_n)$ is the first-order sentence obtained from F by replacing each free variable x_i with t_i. A *boolean first-order query* is a first-order query with arity $n = 0$.

2.1.4 Complexity classes

In this section we briefly sketch the basic notions of computational complexity, and NP-completeness [83, 48]. Given a set of problems A, P^A (NP^A) is the class of problems that are solved in polynomial time by a deterministic (nondeterministic) Turing machine using an oracle for A, i.e., that solves in constant time any problem in A. Furthermore, co-A is the class of problems that are complement of a problem in A. The classes Σ_k^p and Π_k^p of the *Polynomial Hierarchy* are defined as follows:

[2] With the symbol $\not\models$ we indicate the fact that two elements are not in relation

[3] With $\eta[d/x]$ we denote an assign function that behaves like η except for the variable x, where it assumes the value d

$$\Sigma_0^p = \Pi_0^p = P \text{ and for all } k \geq 1, \Sigma_k^p = NP^{\Sigma_{k-1}^p} \text{ and } \Pi_k^p = \text{co-}\Sigma_k^p$$

In particular $\Sigma_2^p = NP^{NP}$, and $\Pi_2^p = \text{co-}\Sigma_2^p$, i.e., Σ_2^p is the class of problems that are solved in polynomial time by a nondeterministic Turing machine that uses an NP-oracle, and Π_2^p is the class of problems that are complement of a problem in Σ_2^p. Finally, PSPACE (NPSPACE) is the class of problems that can be solved by a deterministic (nondeterministic) Turing machine that uses a polynomially bounded amount of memory. By Savitch's theorem [88] it follows that PSPACE=NPSPACE (the theorem actually applies to a much more general class of space bounds).

It is possible to identify three kinds of complexity [94]:

- the *data complexity*, which is the complexity with respect to the size of the underlying database instance, i.e., when the relational query and the schema are considered fixed whereas the database instance is considered as an input to the problem, and
- the *combined complexity*, which is the complexity with respect to the size of the database instance, the query and the schema, i.e., when the query, the schema and the database are considered as the input to the problem
- the *query complexity*, which is the complexity with respect to the query and the schema (but not the database).

In the present work we will mainly refer to data complexity.

2.2 A Framework for Semantic Data Integration

In this section we propose a formalization of a data integration system in terms of syntax (Section 2.2.1) and semantics (Section 2.2.2).

2.2.1 Syntax

A data integration system provides the user with a unified virtual view of data residing at different sources. The layer accessible by the user of the system is called *global schema* and contains a description of the domain of interest, possibly enriched with a set of integrity constraints expressed over global entities. In turn, sources data are transparently accessed by the system through an internal representation of it, called *source schema*. As in general a data integration system has to deal with a variety of heterogeneous data sources, the situation may arise in which the language used to describe the sources within the system is different from the source-specific language. In these cases we can assume to have suitable wrappers that enable the communication with sources, using different languages to describe their data.

Several formalisms have been used in the literature in order to describe both global and source entities in which schemas can be relational [49], object-oriented [9], based on Description Logics [66, 25], semi-structured [78].

The relationships between elements of the global and the source schema are expressed by the *mapping*. The system uses the mapping to access the sources in order to retrieve global data that are relevant to provide answers to the user queries. Two main approaches have been proposed to represent the mapping: *global-as-view (GAV)* and *local-as-view (LAV)*. In the former, elements of the global schema are represented by means of queries expressed over the source schema. Vice versa, in the latter, elements of the source schema are defined by means of queries over the global schema. Both approaches presents advantages and drawbacks. Query answering in GAV data integration system can be in simple cases performed by means of trivial unfolding techniques, but adding or removing sources to the system may lead to the expensive task of redesigning the mapping. Conversely, while additions and deletions on the source schema are faced in LAV systems by simply providing the definition for the new source element, or removing the corresponding source from the schema, the query answering task may be very hard and in general corresponds to a form of reasoning in the presence of incomplete information.

From the discussion above, it follows that the factors affecting the specification of a data integration system are:

- the language used to describe the global and the source schema;
- the formalism adopted for the mapping and the language used to specify queries in the mapping assertions.

In the following we adopt the relational model (Section 2.1.1) for the global and source schema, and use union of conjunctive queries (Section 2.1.2) to define the mapping.

Formally, a *data integration system* $\mathcal{I}$ is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ where:

- $\mathcal{G} = \langle \mathcal{R}_\mathcal{G}, \Sigma \rangle$ is a relational global schema in which $\mathcal{R}_\mathcal{G} = (g_1, \ldots, g_n)$ is a set of global relations and Σ is a set of integrity constraints comprising key, exclusion, and inclusion constraints.
- $\mathcal{S} = \langle \mathcal{R}_\mathcal{S}, \emptyset \rangle$ is a relational source schema in which $\mathcal{R}_\mathcal{S} = (s_1, \ldots, s_m)$ is a set of source relations. Note that no integrity constraints are expressed over source relations: indeed, source are autonomous and independent from the system, and constraints possibly expressed on it are enforced by local data management systems.
- $\mathcal{M}$ is the mapping between $\mathcal{G}$ and $\mathcal{S}$: it contains a set of assertions of the form $\langle r_g, q_\mathcal{S} \rangle$ in which $q_\mathcal{S}$ is a union of conjunctive queries over the source schema $\mathcal{S}$ and r_g is a full query over a single global relation. More precisely, we will denote mapping assertions, with expression of the form

$$\langle g(\vec{\mathbf{x}}), conj_{\mathcal{S}1}(\vec{\mathbf{x_1}}, \vec{\mathbf{y_1}}) \vee \ldots \vee conj_{\mathcal{S}k}(\vec{\mathbf{x_k}}, \vec{\mathbf{y_k}}) \rangle.$$

  in which every $conj_{\mathcal{S}i}$, for $i = 1, \ldots, k$, is a conjunction of symbols of $\mathcal{R}_\mathcal{S}$.

We represent users queries posed to the system $\mathcal{I}$ with union of conjunctive queries expressed over $\mathcal{G}$.

Example 2.2.1 Consider a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$. The global schema $\mathcal{G} = \langle \mathcal{R}_\mathcal{G}, \Sigma \rangle$ contains two relations

$$\mathcal{R}_\mathcal{G} = \{\mathsf{musician}(name, band), \mathsf{rock\_band}(name, country)\}$$

with an inclusion dependency expressed on it

$$\Sigma = \{musician[2] \subseteq rock\_band[1]\}$$

stating that a musician has to in a band. The source schema $\mathcal{S} = \langle \mathcal{R}_\mathcal{S}, \emptyset \rangle$ contains three relations

$$\mathcal{R}_\mathcal{S} = \{\mathsf{guitarist}(name, band), \mathsf{singer}(name, band), \mathsf{band}(name, gendre, country)\}$$

and the GAV mapping $\mathcal{M}$ between $\mathcal{G}$ and $\mathcal{S}$ is defined by means of the following assertions:

$$\begin{aligned} \langle \mathsf{musician}(X, Y) \quad &, \quad \mathsf{guitarist}(X, Y) \vee \mathsf{singer}(X, Y) \rangle \\ \langle \mathsf{rock\_band}(X, Y) \quad &, \quad \mathsf{band}(X, 'rock', Y) \rangle \end{aligned}$$

■

2.2.2 Semantics

Informally, the semantics of a data integration system can be given in terms of instances of the relations of the global schema [70]. An instance for a global relation is a set of tuples that have to satisfy the integrity constraints expressed on it, and the mapping. Three different assumptions that can be adopted in order to correctly interpret the relationship between source data and the data satisfying the corresponding portion of the global schema. Under the assumption of *complete* mapping, sources provide a superset of data that satisfy the global schema. Conversely, under the assumption of *sound* mapping sources provide a subset of data that satisfy the global schema, i.e., retrieved data are sound but may result incomplete. When sources data are both sound and complete, we say that the mapping is *exact*.

In order to formally define the semantics of a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, we consider a database instance $\mathcal{D}$ for the source schema $\mathcal{S}$ and an assumption for the mapping $\mathcal{M}$. We denote the assumption of complete, sound and exact mapping respectively with the notations $as(\mathcal{M}) = c$, $as(\mathcal{M}) = s$ and $as(\mathcal{M}) = e$. Starting form a source database $\mathcal{D}$ and an assumption $as(\mathcal{M})$ for the mapping, we denote the *semantics* of $\mathcal{I}$ with respect to $\mathcal{D}$ the set $sem_{as(\mathcal{M})}(\mathcal{I}, \mathcal{D})$ containing all the global databases $\mathcal{B}$ such that: (i) $\mathcal{B}$ is consistent $\mathcal{G}$ and (ii) $\mathcal{B}$ satisfies the assumption $as(\mathcal{M})$ with respect to the source database $\mathcal{D}$. To specify the notion of satisfying the mapping, we say that, for each mapping assertion $\langle r_g, q_\mathcal{S} \rangle$ and for each assumption $as(\mathcal{M})$, $\mathcal{B}$ satisfies $as(\mathcal{M})$ if:

- $g^\mathcal{B} \subseteq {q_\mathcal{S}}^\mathcal{D}$ if $as(\mathcal{M}) = c$;
- $g^\mathcal{B} \supseteq {q_\mathcal{S}}^\mathcal{D}$ if $as(\mathcal{M}) = s$;
- $g^\mathcal{B} \equiv {q_\mathcal{S}}^\mathcal{D}$ if $as(\mathcal{M}) = e$;

Given a query q of arity n expressed over $\mathcal{I}$, the semantics of q with respect to $\mathcal{D}$ and $as(\mathcal{M})$ is the set of the *certain answers* of q defined as

$$ans_{as(\mathcal{M})}(q, \mathcal{I}, \mathcal{D}) = \{\langle c_1, \ldots, c_n\rangle \mid \langle c_1, \ldots, c_n\rangle \in q^{\mathcal{B}} \text{ for each } \mathcal{B} \in sem_{as(\mathcal{M})}(\mathcal{I}, \mathcal{D})\}$$

Often the problem of *query answering*, that is, the problem of computing the set $ans_{as(\mathcal{M})}(q, \mathcal{ID})$, is addressed by means of *query rewriting* techniques in which the query q over $\mathcal{G}$ is translated into another query q_r called *rewriting*, that can be directly evaluated over the source database $\mathcal{D}$. A rewriting q_r of a query q is a *perfect rewriting* of q with respect to $\mathcal{I}$ and $as(\mathcal{M})$ if

$${q_r}^{\mathcal{D}} = ans_{as(\mathcal{M})}(q, \mathcal{I}, \mathcal{D})$$

for each source database $\mathcal{D}$.

Example 2.2.2 Let us consider again the data integration system presented in Example 2.2.1. Assume to have a possibly inconsistent source database $\mathcal{D}$ containing the following facts:

$$\begin{array}{l}\mathsf{guitarist}(luckater, TOTO)\\ \mathsf{singer}(luckater, U2)\\ \mathsf{singer}(townsend, WHO)\\ \mathsf{band}(TOTO, rock, USA)\end{array}$$

in which $luckater, bono, townsend, TOTO, WHO, U2, USA$ and $rock$ are constant symbols of $\mathcal{U}$. Let us now consider the semantics of $\mathcal{I}$ under the complete, exact and sound assumptions on the mapping $\mathcal{M}$ of $\mathcal{I}$:

- with the *complete* assumption on the mapping $\mathcal{M}$, a global database for $\mathcal{I}$ is every subset of $\mathcal{D}$ that is consistent with respect $\mathcal{G}$ and mapping. In this case we have three possible global database, i.e., $sem_c(\mathcal{I}, \mathcal{D}) = \{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3\}$ where:

  $$\begin{array}{lcl}\mathcal{B}_1 & = & \emptyset\\ \mathcal{B}_2 & = & \{\mathsf{rock\_band}(TOTO, USA)\}\\ \mathcal{B}_3 & = & \{\mathsf{guitarist}(luckater, TOTO), \mathsf{rock\_band}(TOTO, USA)\}\end{array}$$

  It is important to note that given a query q over $\mathcal{G}$, $ans_c(q, \mathcal{I}, \mathcal{D}) = \emptyset$ since $\emptyset \in sem_c(\mathcal{I}, \mathcal{D})$, that is, in the complete semantics there are no answers to the query q.

- Considering the *exact* assumption on the mapping, no global databases exists that are legal with both the mapping and the global schema $\mathcal{G}$. It follows that $sem_e(\mathcal{I}, \mathcal{D}) = \emptyset$ and obviously $ans_c(q, \mathcal{I}, \mathcal{D}) = \emptyset$.

- With the *sound* assumption on $\mathcal{M}$, $sem_s(\mathcal{I}, \mathcal{D})$ contains infinite global databases of the form

  $$\mathcal{B} \quad = \quad \mathcal{D} \cup \{\mathsf{rock\_band}(WHO, \alpha), \mathsf{rock\_band}(U2, \beta)\}$$

  in which α and β are all the possible pair of constants of the domain $\mathcal{U}$. Considering a query $q(X) \leftarrow \mathsf{rock\_band}(X, Y)$ we have that $ans_s(q, \mathcal{I}, \mathcal{D}) = \{TOTO, WHO, U2\}$.

■

In the following we disregard the complete assumption on the mapping, because it is not well suited to deal complex and expressive scenarios in which, for example, integrity constraints are expressed over the global schema. Finally, we refer to [70, 21] for a deeper analysis of the various semantics for data integration systems mappings.

Chapter 3

State of the Art and Current Trends in Data Integration

Data integration is a research field that has been deeply investigated in the last years. Several aspects have been taken into account in such investigations, ranging from data integration system formalization, to query answering, to source wrapping, also in the presence of limitations in accessing the sources. Among all such aspects, the following ones are particularly interesting for the studies carried out in the present thesis:

- the ability of data integration systems to provide *useful* answers to the queries posed by the users, even in situations in which integrated data may result mutually inconsistent;
- the study of the different architectures on which a data integration system can be based on;
- the application to real integration scenarios of techniques to deal with integrity constraints in such a way that efficiency and feasibility are guaranteed.

The goal of this chapter is to provide a survey of the systems and algorithms proposed in the literature that aim to solve the data integration problem. More in detail, Section 3.1 shows some foundational results focusing on the architecture described in Chapter 1, Section 3.2 presents various works focusing on architectural aspects of data integration systems and Section 3.3 shows some of the latest proposal towards efficient data integration.

3.1 Semantic Data Integration

As already explained in Chapter 2, the most natural way to think about a data integration system consists in a layered architecture in which the global schema, the layer accessible by the user, is linked to the source schema by means of a mapping. The particular form assumed by such a mapping affects the techniques adopted by the system in order to answer the user queries.

This Section focuses on the query processing techniques proposed in the literature.

In GAV systems, the query processing problem has commonly been considered easier than the analogous problem in LAV systems. This is because the task of answering queries in the LAV framework intrinsically leads to reasoning in the presence of incomplete information, whereas in GAV, the same task, in simple but practical cases, can be performed by means of easy unfolding algorithms.

In the following we describe the main data integration systems based on the GAV formalism (Section 3.1.1) and present the most important query answering algorithms proposed for LAV.

3.1.1 GAV Data Integration Systems

As mentioned before, query processing in GAV data integration systems can be performed by means of unfolding algorithms, that is, atoms referring to global relations appearing in the body of a query q, can be substituted with the corresponding view definitions contained in the mapping. Although this process is much easier than query answering in LAV systems, it requires a deep understanding of the relationships occurring between the sources; hence, a big part of the work that has to be done for query reformulation, is done at design time, while defining the mapping views. Indeed, in the first prototypes of data integration systems, the notion of global schema was missing, and was constituted by the set of relations exported by wrappers and mediators [95]. Moreover, no integrity constraints were expressed over the global concepts. In other words, mediators realize an integration layer between the user application and the sources. Actually, in such a scenario, to process (positive) user queries it is sufficient to only take into account the global database that can be obtained by evaluating the views in the mapping over the source database (also called *retrieved global database*). Then, to produce query answers the system has to simply evaluate the user query over the retrieved global database. Note that this process is equivalent to an unfolding algorithm.

Some recent works extended the original GAV framework depicted above, adding integrity constraints over the global relations. In this Section we survey the main systems proposed to deal with GAV data integration scenarios focusing first on the simplified setting, and then on more recent and complex scenarios.

Unfolding-based GAV Systems In this Section we present some of the integration systems that adopt the GAV formalism, whose query processing engine is mainly based on unfolding.

TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) is a joint project of the Stanford University and the Almaden IBM database research group [33]. Its architecture is based on a hierarchy of wrappers and mediators: wrappers extract and convert data from each source into a data model called OEM (Object Exchange Model) while mediators combine and integrate data exported by wrappers or by other mediators. There is not a clear definition of global schema, that is in practice constituted by the set of objects provided by wrappers and mediators. A logical language called MSL (Mediator Specification Language) is used to define mediators: such language is essentially Datalog suitably extended to support OEM objects. OEM is a semistructured and self-describing data model, in which each object has an associated label, a type for the value of the object and a value (or a set of values). User queries are posed in terms

of objects synthesized at a mediator or directly exported by a wrapper. They are expressed in MSL or in a specific query language called LOREL (Lightweight Object Repository Language), an object-oriented extension of SQL. The Mediator Specification Interpreter (MSI) [84, 99] processes the queries, and it is constituted by three main components: the *View Expander*, that produce a logical plan for executing the query, the *Plan Generator*, that convert the logical plan into a physical plan, expressed over the sources and the *Execution engine*, which executes the physical plan and produces the answer. TSIMMIS is also able to query sources with access limitation, by exploiting a more complex Plan Generator.

The Garlic project [30], developed at IBM Almaden Research Center, is composed by two main modules: a middleware layer for query processing and data access software called *Query Services & RunTime System*. The middleware layer adopts an object-oriented data model based on the ODMG standard [31] that allows data from various information sources to be represented uniformly. The global schema is substantially constituted by the union of the local schemas, without integrity constraints expressed on it. The Garlic Data Language (GDL), which is based on the standard Object Definition Language (ODL), is used to export the global objects. Each wrapper describes data at a certain source in the OMDG format and gives descriptions of source capabilities to answer queries in terms of the query plans it supports. The notion of mediator is missing in Garlic, and the corresponding tasks are performed by the wrappers. The *Query Services & RunTime System* produces a query execution plan in which a user query is decomposed in a set of sub-queries sent to the sources, by expanding each involved object with its definition provided by the relative wrapper. Every wrapper translates the sub-queries into the source native query language sending back the answers to the user.

The *Squirrel* System, developed at the University of Colorado [102, 101, 100, 63], provides a GAV framework for data integration based on the notion of *integration mediator*. Integration mediators are active modules that support data integration using a hybrid of virtual and materialized data approaches. The initial approach of Squirrel [102, 100] to data integration does not consider virtual views and adopt some techniques related to data materialization. A key feature of Squirrel mediators, which consist of software components implementing materialized integrated views over multiple sources, is their ability to incrementally maintain the integrated views by relying on the active capabilities of sources. More precisely, at startup the mediator sends to the source databases a specification of the incremental update information needed to maintain the views and expects sources to actively provide such information.

MOMIS [9, 10] is a system developed at the University of Milano and the University of Modena and Reggio Emilia. One of its main features concerns semi-automatic techniques for the extraction and the representation of properties holding in a single source schema (*intra-schema relationships*), or between different source schemas (*inter-schema relationships*), and for schema clustering and integration, to identify candidates to integration and synthesize candidates into an integrated global schema. Such relationships can be both intensional and extensional, both defined by the designer or automatically inferred by the system. The integration process is based on a source independent object-oriented model called ODM_{I^3}, used for modelling data sources (structured or semistructured) in a common way. Each mediator is composed of two modules: the *Global Schema Builder*, and the *Query Manager*: the former which construct the global

schema, and the latter which performs query processing and optimization.

Handling Integrity Constraints Some of the main features missing by the systems described in the previous Section are the ability to deal with integrity constraints expressed over the global schema and the ability to work in the presence of incomplete or inconsistent data sources. Indeed, as shown in [70], solving query processing by means of simple unfolding techniques limits the ability of such systems in dealing with complex integration scenarios, e.g., in the presence of incomplete data sources and integrity constraints on the global schema. In this Section a panoramic of the work providing in-depth study of the problem of query processing in GAV is presented.

The *IBIS system* (Internet-Based Information System) [17] is a tool for the semantic integration of heterogeneous data sources, jointly developed by the University of Rome "La Sapienza" and CM Sistemi. IBIS adopts innovative solutions to deal with all aspects of a complex data integration environment, including source wrapping, limitations on source access, and query answering under integrity constraints. In IBIS the global schema is represented in a relational language and is able to cope with a variety of heterogeneous data sources (data sources on the Web, relational databases, and legacy sources). Wrappers provide relational access to non-relational data sources. As far as we now, IBIS is the first attempt of system allowing for the specification of integrity constraints on the global schema, also considering the presence of some forms of constraints on the source schema (source limitation). More in detail, key and foreign key constraints can be specified on the global schema, and functional dependencies and full-width inclusion dependencies, i.e., inclusions between entire relations, can be specified on the source schema. Query processing in IBIS is separated in three phases: (i) *query expansion* that takes into account the integrity constraints in the global schema; (ii) *unfolding* to obtain a query expressed over the sources; (iii) *query execution* which evaluates the over the retrieved source databases, to produce the answer to the original query. Note that while query unfolding and execution are the standard steps of query processing in GAV data integration systems, the expansion phase makes use of the algorithm presented in [16].

The *DIS@DIS System* [18, 22] (Data Integration System ad Dipartimento di Informatica e Sistemistica) is a prototype system developed by the University of Rome "La Sapienza", implementing a variety of algorithm for data integration in the presence of integrity constraints. Both the global and the source schema are represented by means of a relational language, and the mapping is expressed by (union of) conjunctive queries. With respect to the treatment of integrity constraints, the DIS@DIS system is able to deal with both incomplete data and inconsistent data. More specifically, to deal with incompleteness, DIS@DIS adopts the sound semantics, that we have introduced in Section 2.2.2, in which global databases constructed according to the mapping are interpreted as subsets of the databases that satisfy the global schema. Query processing under the sound semantics provides answers that are logical consequence of the global schema and the information available at the sources. Moreover, in order to also cope with inconsistency with respect to KDs and EDs, the system resorts to the loosely-sound semantics, which will be described in Chapter 6, thus allowing for minimal repairs of the sound global database instances. Query processing under the loosely-sound semantics allows for computation of consis-

tent answers from incomplete and inconsistent data. With respect to the mapping specification, DIS@DIS allows for the design of both GAV and LAV mappings.

3.1.2 Query Processing in LAV

Mappings based on the LAV formalism describe the source relations as views defined over the global schema. This means that query processing can be performed answering a query posed over a database schema using a set of views defined over the same schema. This problem is called *answering queries using views* and has been deeply investigated in the literature. Some of its applications are:

- Query Optimization [32]: the use of materialized views may speed up the process;
- Data Warehousing [61, 90, 98, 55]: a data warehouse can be seen as a set of materialized views, hence query processing can be reduced to query answering using views;
- Database Design [91]: using views helps in separating the physical perspective of data from the logical one;
- Query processing in federated databases [76] and distributed databases [65].

One of the most adopted way to face query processing in LAV systems is by means of *query rewriting*. The input of the query rewriting problem is a query q and a set of views defined over a database schema; the process will produce a new query q', called *rewriting* of q, whose evaluation over the views extension supplies the answers to the original query q. Hence, query answering via query rewriting is divided into two steps: a reformulation step, that produce the rewriting of the user query, and the evaluation step that evaluates the rewriting over the views extension. Note that the rewriting process does not take into account the extension of the views, but only the language used to express the query and the views: this situation may lead to some limitation when attempting to produce the answers for a given query q, due to the particular language adopted.

A more general approach, simply called *query answering using views* [4, 50, 28, 29, 25], takes into account the query and views as well as the data at the sources in computing query answer. Such approach overcomes the limitations enforced by query rewriting using views.

In the following we present the *bucket algorithm*, the *MiniCon algorithm* (an extension of the bucket algorithm) and the *inverse rules algorithm*, the most adopted techniques in LAV data integration systems.

Bucket algorithm and MiniCon Algorithm The bucket algorithm [74, 75] is a query rewriting algorithm which works in the case where the query is a union of conjunctive queries and the views are conjunctive queries.

In order to compute all the rewritings that are contained in the original query, the bucket algorithm, presented in [75], exploits a suitable heuristic for pruning the space of candidate rewritings, i.e. queries that could be rewriting of the original query. The algorithm was proposed in the context of the Information Manifold (IM) system [76], a project developed at AT&T.

IM handles the presence of inclusion and functional dependencies over the global schema and limitations in accessing the sources, and uses conjunctive queries as the language for describing the sources and querying the system.

To compute the rewriting of a query q, the bucket algorithm proceeds as follows:

1. for each atom g in q, create a bucket that contains the views from which tuples of g can be retrieved, i.e., the views whose definition contains an atom to which g can be mapped in a rewriting of the query;

2. consider as candidate rewriting each conjunctive query obtained by combining one view from each bucket, and check by means of a containment algorithm whether such a query is contained in q. If so, the candidate rewriting is added to the answer.

If the candidate rewriting is not contained in q, before discarding it, the algorithm checks if it can be modified by adding comparison predicates in such a way that it is contained in q. The proof that the bucket algorithm generates the maximal contained rewriting when the query language is union of conjunctive queries, is given in [50].

According to [74] the bucket algorithm, in practice, does not miss solutions because of the length of the rewritings it considers, but other results [40, 54, 53] demonstrate that in the presence of functional dependencies and limitations in accessing the sources, union of conjunctive queries does not suffice to obtain the maximal contained rewritings, and one needs to resort to recursive rewritings. We refer the interested reader to [40, 67] for a more detailed treatment of the problem.

Two improved versions of the bucket algorithm are the *MiniCon algorithm* [86] and the *shared-variable bucket algorithm* [80]. In both the algorithms the basic idea is to examine the interaction among variables of the original query and variables in the views, in order to reduce the number of views inserted in the buckets, and hence the number of candidate rewritings to be considered. Experimental results related to the performance of MiniCon show that this algorithm scales very well and outperform the bucket algorithm.

Inverse rules algorithm The *inverse rules algorithm* [40] produces a rewriting (query plan) in time that is polynomial in the size of the query. The algorithm was developed in the context of the Infomaster system [39], an information integration tool developed at Stanford University. The inverse rules algorithm constructs a set of rules that invert the view definitions and provides the system with an inverse mapping which establish how to obtain the data of the global relations from the data residing at the sources. The basic idea is to replace existential variables in the body of each view definition by a Skolem function. Given a non-recursive Datalog query q and a set of view definitions $\mathcal{V}$, the rewriting is the Datalog program consisting of both the query and the inverse rules obtained from $\mathcal{V}$.

Note that in the skolemization phase, we just introduce function symbols in the head of the inverse rules and never introduce a symbol within another, which leads to a finite evaluation process. Since bottom-up evaluation of the rewriting can produce tuples with function symbols, these need to be discarded. A polynomial time procedure to eliminate these function symbols by adding new predicates is described in [39]. It is also shown that the inverse rules algorithm returns a maximally contained rewriting w.r.t. union of conjunctive queries, in time that is polynomial in

the size of the query. Even if the computational cost of constructing the query plan is polynomial, the obtained rewriting contains rules which may cause accessing views that are irrelevant for the query. In [73] it is shown that the problem of eliminating irrelevant rules has exponential time complexity. The inverse rules algorithm can handle also recursive Datalog queries, the presence of functional dependencies over the global schema, or the presence of limitations in accessing the sources, by extending the obtained query plan with other specific rules.

3.2 New Architectures For Data Integration

Some recent developments of infrastructures and algorithms for distributed systems, and also the fast growth of the number of heterogeneous information sources scattered on the web, have steadily affected the research in the very last years. Many studies have been motivated by an inadequacy of the systems based on centralized architectures presented so far to satisfy real integration requests. Moreover, in a continuously changing scenario in which sources may join and leave at any time, the design and maintenance of a global schema and a mapping are too expensive. In this section we present a new direction of the data integration research, the extension of the traditional data integration techniques towards more dynamic and flexible scenarios.

3.2.1 Peer-to-peer Data Management Systems (PDMS)

The peer-to-peer (from now on P2P) paradigm consists of a wide network of interconnected computational *peers*, or information nodes, that cooperate with each other exchanging services and information [11]. Each peer shares a part or all of its resources with the network and also benefits of resources offered by other peers. The success of the P2P architecture is determined by several advantages: scalability, robustness and lack of administration needs are some of the main properties of a P2P system. In general the wealth of such systems grows with the number of participants, that increase the memory for data storage, and the computational power of the whole system.

However, as pointed out in [52], often generic P2P systems do not take care of the semantics of the data exchanged. This is a serious drawback, mostly considering that when the network grows, it becomes hard to predict the location and the quality of the data provided by the system. Therefore, there are several problems about availability and consistency of the services provided by the P2P system. In the same paper, the authors identify the *data placement problem*, that is, how to distribute data and work so as to the full query workload is covered with lower cost under the existing resource constraints.

A *P2P data management system* aims to solve these issues. The main goal of such systems is to provide semantic interoperability in the absence of a global information schema. All the knowledge about the topology and extension of the network resides at the peers. In fact, every peer is interconnected with other peers by means of *coordination formulas* [11], that allow every single peer to exploit acquaintances coming from other peers. A P2P system decomposes the user queries by recursively applying coordination formulas, that may also act as a kind of

constraints for the propagation of updates among the network. In [89] the authors present the *Local Relational Model (LRM)*, a data model specifically designed for P2P data management systems. Each peer has a *local database schema* whose semantics is defined on a local domain (disjoint from the domain of other peers). The information propagates between peers thanks to coordination rules and to relations between domains of different peers. Such relations express overlapping between local databases of different peers, i.e., different constants that represent the same object. In such a scenario, coordination formulas are used, in a declarative fashion, to state semantic inter-dependencies between two different local databases. One of the main peculiarities of such formalism is that there is not a concept of global consistency, but only local consistency at peer level.

Many authors have deeply investigated the issues arising in P2P distributed environments [58, 45]. An interesting topic is how to express the logical interconnections between peers. Many of the sources available on the web, for example, cannot be modelled as relational databases, but rather as *data webs* [45] with a set of *entry points*. Under this perspective, the resource scenario can be seen as a set of data webs that can be modelled together as a *web schema*, i.e., a direct graph in which data webs are nodes, and links between them are arcs. In [45] the authors provide a formalization of web schemas architecture, providing also techniques for query answering: a data web can be queried if it has an entry point, and links between data webs can be used to "navigate" through the web schema. Starting from the user query, the technique proposed produces a *navigational plan*, which is then translated into an extension of relational algebra through the *traverse* operator, that allows to traverse links across different data webs. The authors also point out that the LAV and GAV formalism are inadequate to suitably represent link between nodes: in fact they adopt the GLAV paradigm, a generalization of both the GAV and LAV paradigms, in which mapping assertions are constituted by a pair of queries, one expressed over the global schema and one expressed over the source schema. In that paper it is shown that GLAV mapping reaches the tradeoff limit between expressive power and tractability of query answering. It derives that in these settings the complexity of query answering is polynomial in data complexity.

To conclude this short survey, we underline that in many real integration cases, the adoption of database driven techniques can hinder some lightweight data storage task: these issues are addressed in [58]. The scenario proposed in such paper contemplates a peer schema in which peers hold a set of *peer relations*, to represent its data, together with a set of *stored relations*, analogous to the sources of classical data integration systems. Moreover, every peer holds two kinds of peer mappings to interlink its schema with the schema of other peers: *inclusion and equality mappings*, similar to LAV classical mapping, and *definitional mapping*, analogous to GAV ones. The semantic of the whole PDMS is given based on a data instance, that is an extension of the relation of each peer. The consistency of such an instance depends on the satisfaction of inclusion and equality mappings. The authors show that the query answering problem under such assumptions is in general undecidable, due to cycles among definitional mappings. However, bounding the topology of cycles, a polynomially tractable algorithm can be defined, that is able to chain between different peers mapping to get the answer to the query.

3.2.2 Peer-to-peer Data Integration

All the approaches presented in the previous section are not able to deal with an arbitrary topology of peer interconnections, due to the particular semantics adopted in defining query answering techniques. Nevertheless, imposing limitations on peer interconnections does not seem to be a valid approach, since in a dynamic environment, topology is often outside the control of the system itself. The results presented in [26] represent a possible valid solution to that problem. The authors underline that assigning a global semantic to a distributed environment, leads to undecidability of query answering. This is shown in very simple settings, in which only three peers operate. The solution proposed consists in the adoption of an epistemic semantics for the system instead of the classical first order logic semantic. Under this new semantics, mapping assertions, expressed in GLAV, are interpreted so that only the knowledge of the peer is transferred to other peers. In the same work, the authors define a distributed algorithm for query answering that exploits a transaction mechanism to ensure the semantic correctness of the whole process with respect to the epistemic semantics.

The previous work has been then extended in [27] where an infrastructure of P2P data integration, implemented on Data Grids, has been developed. The deployment proposed in this paper models a P2P system as a set of *data peers* and *hyper peers*. The formers are systems that provide data in terms of an exported schema, while the latter do not export data, but are interconnected both with data and hyper peers, building up the topology of the network. An hyper peer connected to other data peers correspond to a classical GLAV data integration system, and its semantics is defined resorting to epistemic logic [26]. Query answering in the Hyper framework is performed by splitting each GLAV mapping assertion into two halves: a LAV assertion and a GAV assertion, linked by means of a new relational symbol. Such LAV assertions are used to produce a logic program exploiting algorithms for query answering using views, while the GAV assertions are used by the system to generate the extensions on which the logic program can be evaluated.

As a further extension of P2P semantic data integration is provided by the same authors in [24]: in that paper an epistemic multimodal logic is adopted and a first treatment of integrity constraints is provided.

3.2.3 Other approaches to Data Integration

To conclude this section, we mention some other approaches to the problem of Data Integration. The *what-to-ask (WTA)* problem, introduced in [23] is the problem of answering the queries posed to a P2P system by only relying on the query answering services available at the various peers. In particular, the setting proposed in [23] contemplates only two peers: the local and the remote peer. That work presents a formalization for the WTA problem in such a framework, providing also a solution for that problem when a basic ontology language is used to express the knowledge base of the two peers. Roughly speaking, a solution for the WTA problem consists in computing the queries that the local peer has to ask to the remote peer in order to answer queries posed to the local peer itself. Moreover, the authors show that, when the expressive power of the ontology language is enriched, the WTA problem is not always solvable. The algorithm proposed

can be seen as a new technique for query rewriting that can be exploited also for data integration systems.

The data integration approach aims to answer the user queries taking advantage of several query rewriting techniques. This means that the data are only at the sources and the system is not in charge to materialize them locally (virtual data integration). Conversely, *data exchange* [44] solutions adopt a strategy of materialization of the remote data. In data exchange we have a remote source schema, i.e., a set of information sources, and a target schema, analogous to the global schema of data integration systems, that has to be materialized. The data exchange problem consists in finding an instance of the target schema, starting from an instance of the remote source schema. Such a problem, that has in general several solutions, is addressed in [43], where the authors presents a formalization of the semantic and query answering for the problem. They define the *universal solution*, i.e., a solution that is homomorphic with all the other solutions, and hence is the most general solution. However, since there can be more than one universal solution, a minimality criterion is used to find the best one. It is possible to identify a substructure common to all universal solutions that is also isomorphic to them. This substructure, called the *core*, can be computed exploiting well-known algorithm of graph theory that, in the data exchange setting, can be executed in polynomial time in data complexity.

3.3 Efficient Query Answering Under Integrity Constraints

Some of the works presented so far pointed out that answering queries in data integration systems under integrity constraints is a very hard task: indeed, also in very simplified settings such problem is untractable, or, in many cases, undecidable.

Some recent works have clearly stated complexity results in that area. In [19], the author analyzed the problem of querying a database that is inconsistent with respect to a set of integrity constraint expressed on its schema, also in the presence of incomplete information. The same authors extended such results to a data integration framework [20], showing that answering queries under keys and foreign keys constraints is in general undecidable, and coNP-hard only for particular configurations of such constraints.

It is important to note that many theoretical issues concerning consistent query answering in data integration systems under integrity constraints can be studied in a single-database settings in which data may not satisfy a given set of integrity constraints. Moreover, due to its wide applicability range, it would be interesting to investigate efficient techniques for consistent query answering, whose aim is finding algorithms and techniques applicable on real cases, that are computationally tractable. In this section we present some recent studies in that area.

3.3.1 First-order Query Rewriting for Inconsistent Database

In order to make query answering easier, one of the possible way amounts to identify classes of query for which the problem is tractable. In [47] the authors identify a class of queries (called $\mathcal{C}_{tree}$) that are first-order rewritable: more in detail, given a database schema with key

constraints expressed on it and a possibly inconsistent database instance, a query rewriting algorithm is provided for consistently querying such an inconsistent instance. The rewriting process produces a first-order query that, evaluated over the original database instance, retrieves only the consistent answers with respect the set of integrity constraints. The advantage of such a technique is twofold: first order queries can be (i) evaluated very efficiently over a database (LogSpace), and (ii) expressed in SQL and thus evaluated using standard database technology. The rewriting algorithm presented in [47] is based on the notion of join graph of a query (see Chapter 6 for details): if the join graph associated to the query is a forest, then the query is first-order rewritable. It follows that first-order rewriting is possible only for a class of queries respecting a condition on the joins. However, the authors show that most of the queries arising in practice are in the $\mathcal{C}_{tree}$ class: for example, 20 out of 22 queries in the TPC-H decision support benchmark [1] are in that class. As a further result, a dichotomy is given for a subclass of $\mathcal{C}_{tree}$: the problem of computing the consistent answers is coNP-complete for every query in that class whose join graph is not a forest.

The feasibility of the first-order approach presented above, has been tested by the same authors in the *ConQuer system* [46]. ConQuer is a system for efficient and scalable answering of SQL queries over inconsistent databases, that allows for the specification of a set of key constraints together with the queries. The system produces an SQL rewriting that retrieves all data that are consistent with respect to the constraints. As a novel contribution, the rewriting takes into account not only Select-Project-Join queries, but also SQL's bag semantics and queries with aggregation: this is an important features ensuring good performance and applicability in many practical cases. In the same work, a further rewriting technique is presented that makes use of annotations indicating which tuples are known to be consistent. Such annotations can be exploited to perform additional optimizations unenforced by standard query optimizers. Some experimental result are also provided showing the overhead of resolving inconsistency at query time over databases with an high number of inconsistencies.

3.3.2 Querying Inconsistent Database through Probabilistic approaches

The presence of duplicated tuples is one of the most likely situations arising when integrating data coming from different sources. Also when sources are known to be consistent with respect to integrity constraints expressed on it, some violations may occur when merging their data into a single database. Many techniques have been proposed to identify such tuples, but the elimination of it can be a difficult task that often relies on manual and ad-hoc solutions. In the context of the ConQuer [6] system, a complementary approach has been proposed that permits declarative query answering over database with duplicated data. The main idea is to associate every tuple with its probability to be in the clean database. To this aim the authors present a new semantics that uses the notion of *clean answer*. Such semantic is independent of the way the probabilities are produced, but is able to effectively exploit them during the query answering process. The first attempt of considering tuples probabilities during query answering was presented in [38]: in that work, the authors consider a semantic in which tuple probabilities are independent, that is,

the probability of a tuple to be in the clean database is independent from the one of any other tuple. In the semantics of clean answers presented in [6] the authors consider every possible database obtainable starting from the inconsistent one, by choosing exactly one tuple for each duplicate. Moreover, given two conflicting tuples t_1 and t_2, if t_1 is in a possible database, than the probability that t_2 is in the same database equals to zero. It follows that tuples probabilities are not independent. According to the semantics of clean answers, a rewriting technique is presented, working for a large subclass of Select-Project-Join queries, which produces an SQL query used to compute the answers to queries in that class. Different probabilities are also assigned to duplicated tuples exploiting a tuple matching technique. Some experimental results show that rewritten queries can be efficiently evaluated over large databases, without introducing a significant overhead on the execution time of the original query.

3.3.3 Other approaches to Efficient Query Answering

In this section we present some other works in the field of efficient query answering. In [97] the authors propose a technique to deal with violations of functional dependencies, that works by rectifying the original database before proceeding to query answering. The *project-join-repair* approach starts from a set of inconsistent relations and exploits joins between them to build a unique, possibly inconsistent, database. After this step, several repairs of that inconsistent database are produced and then queried on the basis of a semantics for incomplete databases. The main result of the paper is the tractability of consistent query answering for a certain configuration of the functional dependencies expressed over the database, and for a class of queries. To this aim the authors define the *rooted rule*, a subclass of conjunctive queries, that together with acyclic functional dependencies, can be evaluated efficiently over the rectified database instance.

Another interesting work in the field of consistent query answering is [36]. In that work the authors define a query answering process based on the notion of *conflict hypergraph*, a compact representation of all the repairs of a given database with respect to some integrity constraints. In particular the authors consider projection-free relational algebra queries with union and set difference, over database schemas enriched with denial constraints. The possibility to handle queries with union is a novel contribution and allows for the extraction of indefinite disjunctive information from an inconsistent database. The authors extended a previous result on the tractability of query answering of ground queries with respect to a set of denial constraints [34], showing that an analogous complexity result holds for quantifier-free relational calculus queries. The evaluation process presented in [36] is structured as follows: (i) before processing the query, a conflict detection is performed over the database, and the conflict hypergraph is produced; (ii) the original query, expressed in projection-free relational algebra, is translated into a quantifier-free first-order formula; (iii) such first-order formula is then grounded with an appropriate set of bindings for its variables; (iv) the grounded formula thus obtained and the conflict hypergraph are sent to *HProver* [34], and thus evaluated in polynomial time. The whole query answering process summarized above is implemented in the context of the *Hippo* [36] system, a prototype that uses a relational DBMS as a back-end for storing data. The output of the system are the

consistent answers to the input query.

Chapter 4

Commercial Systems for Data Integration

Enterprise integration is one of the most important keywords of the current IT scenario. In the context of enterprise integration, data integration is a crucial need enforced by a number of enterprises and organizations. The need of integrating and sharing data residing at different sources emerges both at inter-enterprise and intra-enterprise level. Indeed, it is a common situation that different parts of the same organization adopt different tools for data storage. This diversity is due to a number of factors: geographic separation of work groups, diverse adoption rate of new technologies, different dimensions of business entities, mergers and acquisition of smaller enterprises, are the most frequent ones. Furthermore, different organizations show increasing needs in sharing data with other business partners.

Many software vendors provide different solutions for such kind of problems that have been historically faced through database replication systems or data warehouses. However, in the last years, IT systems have been called upon to support information integration in real time so that up-to-date information are always available and ready to use. Some of the central features provided by major software companies concern distributed database systems, efficient technologies for data search and the ability to access a number of heterogeneous data sources.

In this chapter we will analyze three comprehensive solutions to the information integration requirements coming from leading software producers: Oracle, IBM and Microsoft. It is well known that such companies provide largely adopted solutions for data storage: Oracle DBMS [82], SQL Server [79] and DB2 Universal Database [64] are three of the most popular database management systems available on the market. In the latest versions of such products, some solutions to the data integration problem are also provided. We will present the main features of Oracle 10g Information Integration, Microsoft SQL Server 2005 Integration Service and IBM DB2 Information Integrator respectively in sections 4.1, 4.2 and 4.3. It will come out that a discrepancy arises between the meanings that the term *data integration* assumes in the commercial environment, and the meanings it assumes in the corresponding computer science research area. We will point out some limitations enforced by commercial tools for data integration in Section 5.1, investigating

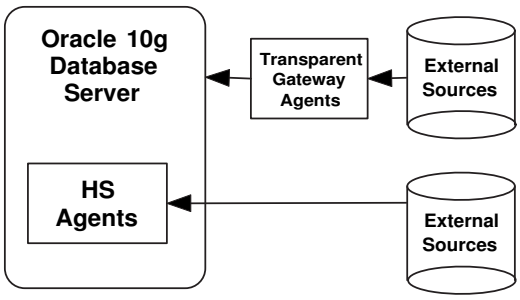


Figure 4.1: The Oracle 10g Information Integrator Architecture

possible ways to overcome such limitations.

4.1 Oracle 10g Information Integration

To meet the latest requirements for information integration, Oracle provides *Generic Connectivity* and *Transparent Gateways*, the main features of *Heterogeneous Service* (HS), and new software modules called *agents* enabling communication with non-Oracle systems. Oracle8 HS is integrated in the Oracle database server and extends the Oracle SQL Engine in order to recognize the capabilities of the remote non-Oracle system. Together with the ability to connect other systems, also data-type translation and data dictionary (database catalog) mappings are provided. As part of the Oracle server, HS takes advantage of its SQL parsing and distributed optimization capabilities, also preserving transactions coordination between Oracle and the connected non-Oracle system by means of the two-phase commit protocol provided by the Oracle server itself. More in details, the following services are provided by HS:

- *Transaction service*, which enables the Oracle environment to manage authenticated sessions with other systems. Once the connection is established, the non-Oracle system is transparently accessed and the corresponding connection closed when the Oracle client session is complete. Global data integrity is also preserved even if the actual non-Oracle system does not provide two-phase commit protocol support.

- *SQL Service* which provides the translation capabilities needed when interacting with different system: (i) *SQL Translation* that translates Oracle SQL language in the specific non-Oracle system dialect, and (ii) *Data Dictionary Translation* that provides an homogeneous way to querying the remote database catalog.

- *Procedural Service* which is a programmatic interface for executing stored procedures on a non-Oracle system.

- *Pass-through SQL* which allows for issuing native SQL queries on a remote system, even if it is not supported by Transparent Gateways and Heterogeneous Service.

Agents are the interface to non-Oracle systems (see Figure 4.1) providing SQL Translation and data type conversion. They interact with HS to enable connection transparency. Additional

agents may be added when the heterogeneous environment grows. Oracle agents can be divided in two categories: *HS agents* and *Transparent Gateways agents*. The former can be of two types based on the specific driver they utilize (ODBC driver and OLE DB driver), and are used by Generic Connectivity to access specific data sources, such as flat text files, Excel files and many others for which an ODBC (or OLE DB) driver is available. Transparent Gateway agent are separate from the Oracle server: differently from the HS agents, that must be installed on the same machine, they can be installed on the remote machine. When this happens, agents use the non-Oracle system native interface in order to achieve maximum performance and functionality. Along with the above mentioned SQL service, Transparent Gateways agents also provide the *server isolation* feature: it runs as a separate process from the Oracle server protecting it by potential failures and other problems related to the non-Oracle libraries.

Several levels of transparency are achieved by HS:

- *Operational transparency*: the target non-Oracle system looks like a remote Oracle server since agents provide suitable translation for SQL, Data Dictionary and data type to the specific dialect of the target and secure transaction managing based on the two-phase commit protocol. This consistent interface decreases development time of distributed applications also increasing its transportability.

- *Location transparency* can be achieved by defining views or synonyms for each remote object, so as it appears as a local object.

From a practical point of view, the access to an external data source can be accomplished by the following steps: (i) a new ODBC data source name (or OLE DB connection) must be added to the system data source names for the specific source being accessed; (ii) a *listener* for the new created data source name has to be added to the Oracle Server by creating a text file with the necessary information within the Oracle installation directory; (iii) a new *service* must be declared responsible for the interaction with the listener specified in the previous step by editing an Oracle configuration file; (iv) a *database link* can be created to establish the connection with the data source. Often ODBC data source names can be used to access a number of resources residing in the same location (for example, a set of flat text files in the same directory, or different spreadsheets of the same Excel workbook). Hence, single resources residing at the same database link can be referred with the following syntax: `resource_name@database_link`. In order to write queries in an easier way, *synonyms* can be created so as to access remote data sources as if it were local tables.

As a final remark we remind that Oracle 10g Database Server provides XML native support. XML documents can be stored in relational tables (see Example 4.1.1 for details) using the appropriate built-in data type. When inserting XML data into a table, an XML schema can be provided for validation purposes. Together with the Heterogeneous Service, the XML support gives Oracle the ability to handle a variety of heterogeneous data sources as if they were a single database.

Example 4.1.1 The aim of this example is to show how XML data can be stored in and retrieved from an Oracle table. Suppose that XML documents are stored in the local file system directory

`c:\data\xml`, and assume that the document we are interested in is the `university.xml` file. A fragment of that document is shown below:

```
<universities>
    <university code="000">
        <city name="ROME"/>
        <name>LA SAPIENZA</name>
    </university>
</universities>
```

First we have to create a reference (named *xmldir*) to the specific directory containing XML documents:

```
CREATE DIRECTORY xmldir AS 'c:\data\xml';
```

Second, we have to create a table for storing the documents. This can be done by means of the following Oracle's DDL statement:

```
CREATE TABLE university_xml (xmlfield XMLTYPE);
```

Now we are ready to insert the document into the database table:

```
INSERT INTO university_xml values
(
    XMLType
    (
        bfilename('xmldir','university.xml'),
        nls_charset_id('AL32UTF8')
    )
);
```

Note that the insert statement makes use of the `XMLType` function, that parses and validate the XML data being inserted. We conclude this example showing a query that retrieves the data stored into the `university_xml` table.

```
select
    extractvalue(value(m), '/university/@code'),
    extractvalue(value(n), '/city/@name'),
    extractvalue(value(m), '/university/name')
from
    university_xml_table,
    table(xmlsequence(extract(field1, '/universities/university'))) m,
    table(xmlsequence(extract(value(m), '/university/city'))) n;
```

■

4.2 Microsoft SQL Server 2005 Integration Service

Since its previous version (SQL Server 2000) SQL Server, the Microsoft database management system, offers some data integration related features. The *Data Transformation Service* (DTS) available in SQL Server 2000, has been improved by the new *SQL Server Integration Service* (SSIS) platform, based on the *Extraction Transformation and Loading* (ETL) technology. SSIS enables organizations to easily integrate data coming from heterogeneous information sources by providing a number of features and high-scale performance necessary to build effective information integration.

The concept of *package* (inherited from the previous version of SQL Server) resides at the heart of the integration services and represents a unit of work, in the integration plan, that can be independently saved, retrieved or executed: it also serves as a container for other objects that take part to the whole process. Two main categories of objects can be distinguished within a package, *control flow* objects and *data flow* objects. The former determine processing sequences and consist of:

- *Containers*: groups of interconnected operations varying from other containers to packages. The workflow is determined by the type of the specific container: Sequence, For Loop, ForEach Loop, TaskHost and EventHandler. Containers can be embedded in a modular way to achieve complex control flows.

- *Tasks* are responsible for performing the actual work. Its main feature is data retrieval across a variety of repositories through the interaction with packages. Tasks can be divided into a number of categories: Data Flow tasks, Data Preparation tasks, SQL Server Tasks, Analysis tasks and other. Although already present in SQL Server 2000, the number of tasks available in SQL Server 2005 Integration Service has increased almost twofold.

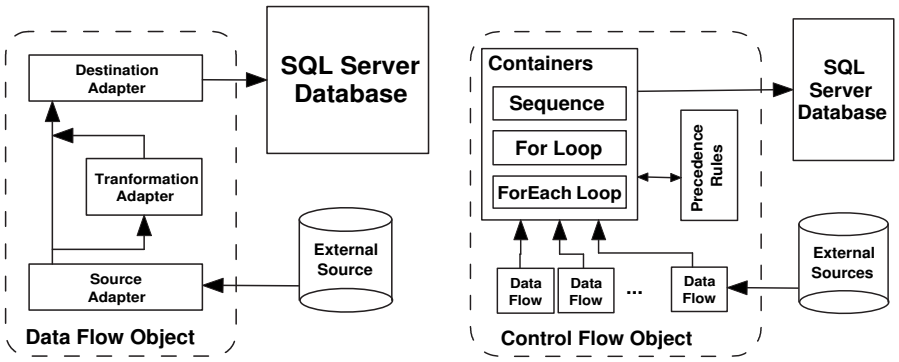
- *Precedence constraints*: determine the sequence in which links between task and containers are processed by providing transitions from a task or container to another. By means of constraint properties, the conditions that determine if a transition will occur are evaluated, also on the basis of the execution status (success, completion, failure) of preceding constraints.

The latter determine the flow of data and can be further categorized in:

- *Source adapters*: permit the extraction of sources data (flat files, Excel files, OLE DB connections, generic ODBC connections). Some scripting facilities are also included to allow for non built-in functions to be implemented and executed. Script components can also be used to implement transformation and destination adapters.

- *Transformation adapters* allow modification of data by means of aggregation functions and data type conversions.

- *Destination adapters* load output of data flows into target stores. Target stores can be flat files, Excel files or generic databases accessible via various connections (OLE DB, ODBC).



(a) Data Flow Objects (b) Control Flows Objects

Figure 4.2: SQL Server Integration Service Architecture

Commonly, data flow tasks start from a source adapter, passing through transformation adapters towards destination adapters.

Data and control flow elements also include other types of components such as *connection managers* and *variables*: the first ones work as logical representations of connections with different data sources along with their properties, while variables are a temporary storage for parameters whose values can change during the whole process. Together with these new types of components, also *event handlers* and *log providers* have been added as new features in SQL Server 2005. Event handlers allow for custom pieces of code to be invoked in response to a specific event triggered by containers. Log providers capture the sequence of event taking place during tasks execution.

Along with the improvement of the DTS functionalities, also built-in XML support is provided by SQL Server 2005. Together with relational data, also XML fragments can be stored in standard tables, by means of the *XMLType* native data type. It follows that semi-structured data can be stored, queried and merged with pre-existing database tables. Once stored into an XMLType-typed column, XML information can be automatically parsed at query-time and transformed into an equivalent relational representation. A collection of XML schemas can be added to the database catalog, and occasionally used to validate XML fragments being stored into the database. The way XML data can be stored and queried in SQL Server 2005 is similar to the one presented in Example 4.1.1 for the Oracle system.

4.3 IBM DB2 Information Integrator

The DB2 Information Integrator (DB2II) [3, 2, 57, 56, 15] architecture provides a set of new functions and objects whose aim consists in providing uniform access to a number of heterogeneous data sources, as if it were contained within a local database instance. DB2II provides a set of *wrappers* enabling the federation of a number of data sources. In the IBM's terminology, a wrapper is a library that allows access to a particular type of data source or protocol, contains information about the remote data source characteristic and understands its capabilities. The

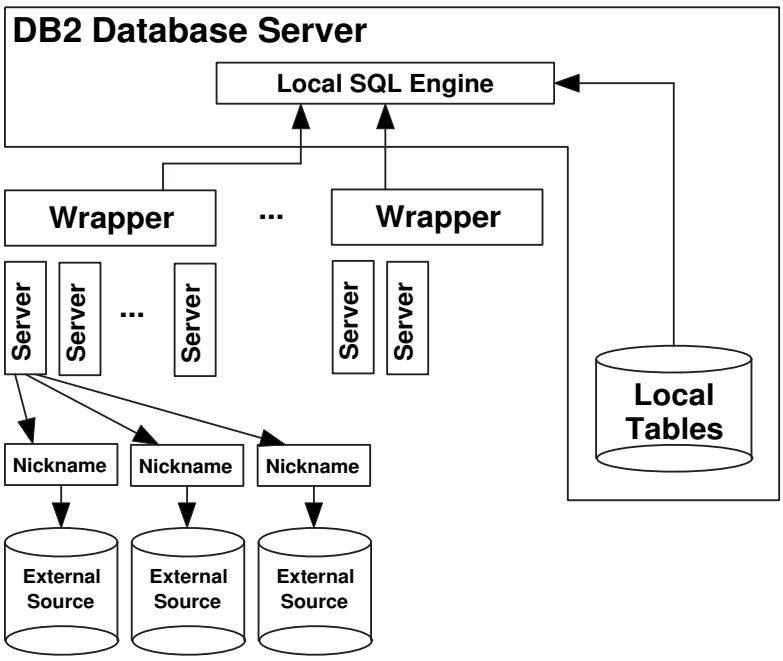


Figure 4.3: The DB2II Architecture

DB2II's wrappers family is divided in two categories: *relational wrappers* enabling access to others' vendor DBMS (Oracle, Sybase, Microsoft SQL Server, Teradata and generic ODBC Sources) and *non-relational wrappers* allowing access to a variety of non-relational data sources (flat files, Microsoft Excel files, XML files, Extended Search, Documentrum and BLAST data source). A wrapper must be instantiated to gain access to one or more data sources of the corresponding type. To allow the federation of more than one source of the same type, the creation of a *server* for each source is needed. A server represents a specific data source that is accessed through a wrapper. For every remote server, one or more data objects (mapped to rows and columns) can be referred through *nicknames*, i.e. aliases that enable querying of remote data as if it were local tables. The complete architecture of DB2II is depicted in Figure 4.3. For each data source requiring an account to access its data, *user mappings* must be created in order to associate a local user ID with the remote user ID and password needed to access source data. The federation of a data source, i.e. the creation of the corresponding wrapper, server and nickname, can be performed easily with a set of queries expressed by means of a DB2II SQL dialect (see Example 4.3.1).

Often when querying remote relational data sources, a discrepancy may arise between the data type returned by the sources and local DB2 data type. The default mappings are in the wrapper library, but additional semantic definition may be given by *data type mappings*. Furthermore, the wrapper library also contains default *function mappings* between DB2 built-in functions and the relational data source equivalent. If needed, additional user function mappings may be defined in case of pushing down a function to the data source that does not have a corresponding function in DB2.

A DB2 database server with DB2II installed on is referred to as a *federated server*. The DB2 instance that manages federated sources is called a server because it satisfies requests from users and applications. Often a federated server sends part of the users' requests directly to the data sources: the so called *pushdown* operation allows for load-balancing and global workload optimization. Some of the main features provided by a federated server are:

- *Transparency*: differences, idiosyncrasies and implementation of the underlying data sources are masked to the user;
- *Heterogeneity*: the ability to federate diverse type of data sources, both relational and non-relational;
- *Extensibility*: it is easy to add (or remove) new data sources to the federated database schema;
- *Autonomy*: data sources can be federated with no impact on existing applications.

Example 4.3.1 As an example we show below the steps that have to be performed in order to federate an XML data source. Let us consider the following fragment of an XML document:

```
<customer>
    <name>Mario Rossi</name>
    <address>Via Salaria, 113</address>
    <city>Rome</city>
</customer>
```

Suppose we want to access such document through DB2II federation technology. First we have to instantiate the wrapper for XML data source files:

```
CREATE WRAPPER xml_wrapper LIBRARY 'libdbxml.dll';
```

Once a wrapper has been created, we must declare the server that will connect to the specific data source:

```
CREATE SERVER xmlserver WRAPPER xml_wrapper;
```

We are now able to define the nickname allowing us to interact with the XML document:

```
CREATE NICKNAME customer (
    name VAHCHAR(30) OPTIONS(XPATH './name/text()),
    address VAHCHAR(50) OPTIONS(XPATH './address/text()),
    city VAHCHAR(30) OPTIONS(XPATH './city/text())
)
FOR SERVER "xmlserver"
OPTIONS(XPATH '//customer', FILE_PATH '/customer.xml')
```

Other types of sources can be federated in a similar way. ∎

Oracle 10g Information Integration	SQL Server Integration Service	DB2 Information Integrator
SQL Server	Oracle	Oracle
DB2	DB2	SQL Server
Sybase	Exchange	Sybase
Informix	Access	Teradata
Teradata	Excel	Documentrum
Ingres	ODBC	BLAST
ODBC	OLE DB	XML
OLE DB	XML	Extended Search
		ODBC

Figure 4.4: Heterogeneous Data Source Support

4.4 Discussion

In this chapter we presented three commercial systems for information integration coming from three leader software vendors. A report on the systems performances is out of the scope of this chapter, and will be presented in Chapter 7, where detailed experimental results are shown, together with a description of the experiments conducted on the three systems. We conclude with some considerations about the effective use of such systems for data integration purposes. In particular we compare Oracle 10g Information Integrator, SQL Server Integration Service and DB2II on the basis of the following factors: *heterogeneity*, i.e. the number and types of different sources accessible by the respective integration capabilities, *configurability*, that is, how the task of adding new sources to the system can be automatized, *portability*, i.e. the ability of the systems to run in diverse operating systems and platform, and *virtuality*, how much sources retain their independence from the integrated data. Figure 4.5 summarizes such comparison.

Heterogeneity The commercial systems presented so far offer a wide range of source types accessible by their integration services. A detailed list of them can be found in Table 4.4. However, DB2II offers a built-in Software Development Kit that enables users to develop their own wrapper libraries. This can be accomplished also in Oracle and SQL Server by solving the difficult task of writing an appropriate ODBC driver. As a final remark, DB2II offers native wrappers for flat text files and Excel files, that avoid the use of ODBC drivers.

Configurability As shown in Example 4.3.1, the task of adding a new source into the DB2II environment can be done in a very easy way by issuing appropriate SQL-like statement. This allows for a full automatization of such task, a very important feature in scenarios in which new sources are frequently added to the system. This can be achieved also in the Oracle system, although it is worth to note that writing files to the file system (see Section 4.1) requires appropriate rights that can affect to global security of the system. Moreover, adding new sources to the Oracle environment, requires the service responsible for sources interactions to be restarted:

	Heterogeneity	Configurability	Portability	Virtuality
Oracle 10g Information Integration	✓		✓	✓
SQL Server Integration Service	✓			
DB2 Information Integrator	✓	✓	✓	✓

Figure 4.5: Systems Comparison

this could be a problem for systems with a high workload. In SQL Server the task can be performed only through its visual interface and requires human intervention: in our opinion this is a drawback that limits the applicability of the system in many practical cases.

Portability Since its previous versions, both Oracle and DB2 databases are multi-platform systems. This allows for building up information integration solutions on a variety of operating systems and platforms. On the other hand, SQL Server runs only on machines equipped with Microsoft operating systems.

Virtuality Oracle 10g Information Integration and DB2II adopt a fully virtual approach while accessing remote data sources. Sources retain their autonomy and modifications made by third users or applications are always available to the integrated system without having to reload it. Conversely, SQL Server Integration Service physically imports remote data into local data destinations. Although this provides high performance when querying the sources, the system is thus forced to completely reload sources when they change, which is a serious drawback.

Chapter 5

An Efficient System for Data Integration

In this chapter we present an novel approach to solve the data integration problem based on commercial tools. As we will explain better in Section 5.1, commercial tools exhibit some limitations with respect to the requirements of a data integration scenario [85]. Starting from such limitations we devise a way to overcome it, and show (in Section 5.2) the architecture of a system that is able to manage complex integration environments with a number of heterogeneous sources being integrated under a common view of it. The system provides the user with a classical database interface through which the sources are transparently integrated and queried. The independence and autonomy of data sources are also preserved by means of a fully virtual approach.

5.1 Limits of Commercial Systems w.r.t. Semantic Data Integration

Let us consider the approach adopted by the systems presented in Chapter 4. In order to deal with the information integration problem, they mainly act as data federation tool, enabling users to access remote data sources as if it were contained in the local database. However, they do not let the designer define an arbitrary description of the domain of interest. In particular, we have identified two main limitations affecting the effective employability of the systems analyzed:

- external data source can be modelled as aliases that can be queried as if it were local tables. However, the correspondence between remote data sources and local alias is always one-to-one, instead of letting the designer define more expressive correspondence between a virtual concept of interest and, for example, a view or a query over multiple source data sets (as those foreseen by the flexible architecture presented in Section 2.2).
- Furthermore, if the sources are relational, the aliases' schema are identical to those of the modelled data source relations, which means that both have the same number, name and type of attributes. Even if sometimes the definition of aliases is a little more flexible for

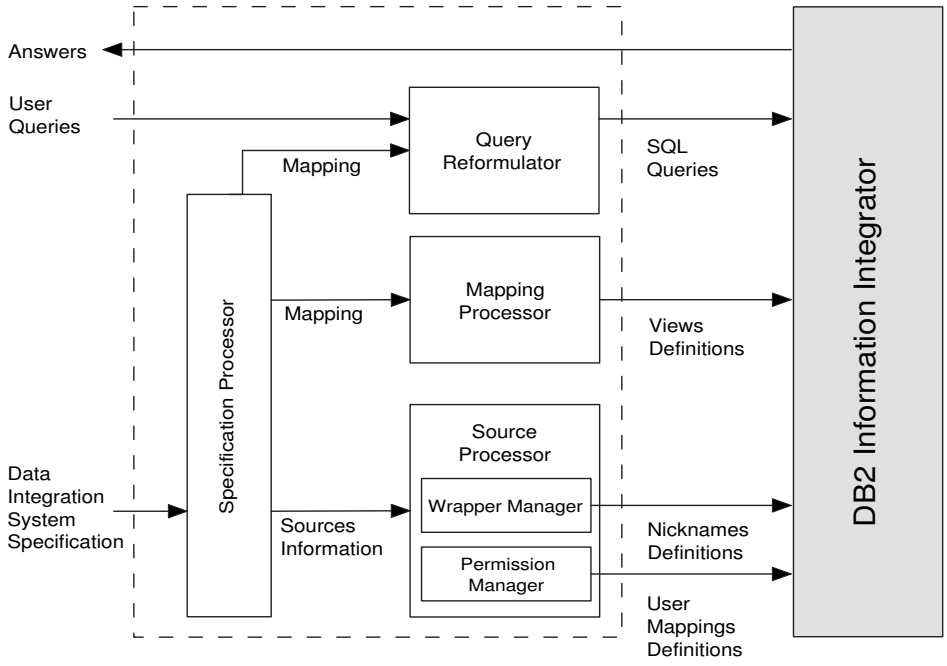


Figure 5.1: System architecture

non-relational data sources, there are still several rules that the designer has to follow, which limit the expressiveness of the correspondence even inside a single source data set.

The above mentioned limits are typical of data federation tools. Indeed, in this kind of tools, the designer is simply provided with a view of data sources that is strongly source-dependent, since it basically reflects the source structure.

5.2 Architecture of the system

In this section we describe the architecture of a novel system for efficient semantic data integration (depicted in Figure 5.1), based on commercial tools for information integration, and explain its behavior. The system can be view as a plug-in that is able to cooperate with the commercial tools described in Chapter 4 in order to achieve effective data integration. In the following we first sketch the overall architecture of the system, and then provide a detailed description of each system module.

Our system takes as input the data integration *system specification* provided by the designer. Such specification contains:

(i) the description of the global schema, i.e., of the relations that are directly accessible by the users;

(ii) the specification of the mappings between global and source schema;

(*iii*) a description of the sources interacting with the system through the mappings.

For each source, some additional information has to be provided in order to enable the access to it. Such source-specific information change on the basis of the particular tool adopted (Oracle, DB2 or SQL Server). In the following we refer to DB2 Information Integrator (DB2II) for technical details about source wrapping, but our description is absolutely general and can be trivially adapted to the use of a different tool.

Informally, the system operates in two modes. The first, called *setup mode*, is performed by the *Specification Processor* module and is enabled when the specification of a new data integration system $\mathcal{I}$ is processed: the system reads the specification provided by the designer, and performs a series of actions in order to set up the integration environment. During this phase, a set of SQL statements are produced that enable DB2 Information Integrator to access the sources; the connection with the sources is ensured by exploiting *nicknames*: a nickname is an alias that allows DB2 Information Integrator to interact with a source as if it were a local relational table. In other words, all the sources appear to be part of a unique federated database (see Section 4.3 for further details). Furthermore, in the setup mode, the mapping assertions (expressed according to the GAV paradigm) are used to produce a set of view definitions which allow us to effectively integrate data stored in different sources. In a nutshell, the final result of the setup mode is a new data integration system specification $\mathcal{I}'$ obtained from $\mathcal{I}$, composed by: (i) a nickname for each element of the source schema and (ii) a view, defined over nicknames, for each element of the global schema.

The second mode is called *run mode*, and is enabled every time a user issues a query to the system. Users' queries are posed over the elements of the global schema, and can be processed by means of two different techniques described in detail in the following (see Section 5.3). More precisely, during the run mode a user query q expressed over a data integration system $\mathcal{I}$ is translated into a new query q' expressed over the compiled specification $\mathcal{I}'$. Figure 5.2 summarizes the behavior of the system.

In Chapter 6 we devise a technique to handle integrity constraints expressed over the global schema.

5.2.1 System Specification

The System Specification is provided by the designer and contains a full textual description of all the components of a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$. It is divided into four sections: *Global Schema* $\mathcal{G}$, *Source Schema* $\mathcal{S}$, *Mapping* $\mathcal{M}$ and some *Sources Information* needed to physically access the sources. The first contains a relational representation of the global relations accessible by users of the data integration system: each global relation is described by its name and the names and types (optional) of its attributes. The second describes the set of sources accessed by this data integration system instance: a relation schema is provided for every source accessed by the system. In the third section a set of GAV mapping assertion are provided in order to establish the correspondence between global and source entities. In details, mapping assertions are expressed by means of union of conjunctive queries whose head refers to a global relation, and whose body is a union of conjunctions of relations of the source schema. The fourth

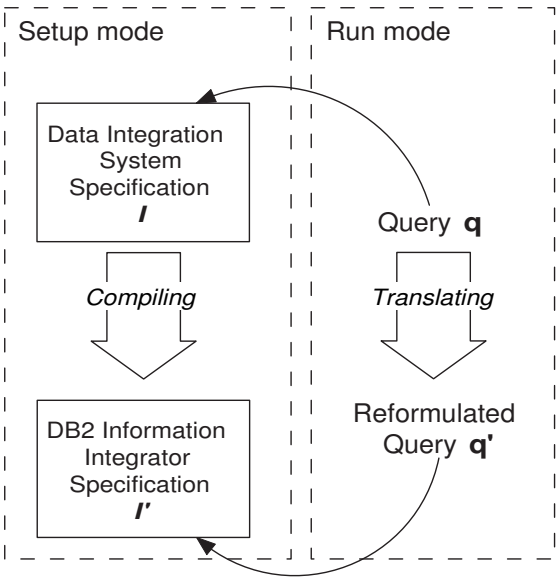


Figure 5.2: The behavior of the system

section provides source-specific information necessary for the system to access the sources and contains a sub-section for every source appearing in the source schema. An example of system specification is presented in Example 5.2.1.

Example 5.2.1 Suppose we want to build a data integration system managing information about soccer game. The system will use data coming from three different sources: an Excel file containing data about soccer teams playing in the Italian championship; a flat text file containing information on soccer teams playing in the Spanish championship; a MySQL database holding information about European football players, for which an ODBC connection is available. We build a global schema with the player and team relations: the first extract information from the MySQL database source, the second extracting data from both the Excel file and the flat text file sources. The system specification for this simple example is shown in Figure 5.3. ∎

5.2.2 Specification Processor

The Specification Processor analyzes the data integration system specification (global schema, sources schema, mapping and sources information) provided by the designer. First of all the module checks for the correctness of the provided system specification: (i) relation symbols occurring in the mapping section have to be declared in the global schema and source schema section; (ii) in the mapping definitions global and source schema relations should reflect the right signature; (iii) the mapping has to be expressed according to the GAV formalism.

Some *type checking* tasks are also performed while setting up the data integration system instance: in order to effectively retrieve meaningful information from multiple sources, joins occurring in the mapping definitions must occur between attributes of the same type. Moreover if

[global schema]	**[s1 source information]**
player(name, team, age)	type=Excel
team(name, leader)	file_name='squadre.xls'
[source schema]	**[s2 source information]**
s1(team_code, name, leader, city)	type=textfile
s2(name, leader, ranking)	file_name='spanish_teams.txt'
s3(player_code, name, team, age)	**[s3 source information]**
[mapping]	type=ODBC
player(X,Y,Z) :- s3(W,X,Y,Z).	dsn=mysql_db
team(X,Y) :- s1(W,X,Y,Z).	username='root'
team(X,Y) :- s2(W,X,Y).	password='elvis'

Figure 5.3: Example Data Integration System Specification

the attributes of the global relations are typed, the system should check that type correspondence is preserved by the mapping when passing from global to source relations. To this aim the Specification Processor performs two checks:

- *Mapping data-type checking* verify that joins in the body of the mapping definitions involve attributes of the same type. For example, let us assume that the query

$$r(X) : -S_1(X, Z), S_2, (Z, Y). \tag{5.1}$$

  is defined into the mapping section. The system will check that the second attribute of relation S_1 is of the same type of the first attribute of relation S_2.

- *Global data-type checking* ensures that if data-type are specified on global relation attributes, such attributes are correctly associated, in the mapping assertion, to others source relation attributes of the same type. Consider again the mapping defined by the query 5.1: the Specification Processor will check that if a type is given for the first attribute of the relation r in the global schema, then such type has to be the same of the first attribute of the source relation S_1. To do so, the module accesses the source's catalog information.

After the type checking tasks the module also extracts information about mapping and sources, passing it respectively to the *Mapping Processor* and to the *Source Processor*. It also stores the meta-data system information in a local meta-data repository, used to retrieve system information for future uses. Furthermore, same sources could be exploited in different data integration systems: all information about wrapped sources will be available for other integration contexts through the meta-data repository.

5.2.3 Sources Processor

In order to access heterogeneous sources, we need to wrap them. To this aim the Source Processor exploits its sub-module called *Wrapper Manager*. This module receives sources' details from the Specification Processor and produces a set of SQL statements that form the input for DB2II. As

already shown in Section 4.3, such SQL statements are expressed in an IBM dialect that allow DB2 Information Integrator to federate the sources and interact with the them as if they were part of a local database. For those types of sources for which some permissions are required (user name and password, for example), the Source Processor exploit its sub-module *Permission Manager* that produces a set of SQL statements for user mappings creation: a *user mapping* is an association between the local DB2 user ID and the remote user ID and password needed to access the source. The outputs of the Permission Manager and Wrapper Manager modules are then passed to DB2II through JDBC in order to realize the connection to the sources. Once a source is connected, the Source Processor tests it with a sample query, in order to ensure that the connection to the source is correctly working.

Example 5.2.2 Consider the specification presented in Example 5.2.1. In order to access the MySQL database the Source Processor accomplishes two task: (i) creates the user mapping between the local DB2 user `db2admin` and the remote MySQL user `root` identified by the password `elvis`; (ii) instantiates the wrapper for ODBC data sources, and creates the DB2's server and nickname definitions (see Section 4.3 for detail). The first task, performed by the Permission Manager, yields the following SQL statements:

```
CREATE USER MAPPING FOR "db2admin" SERVER "odbc_server"
OPTIONS
(
    ADD REMOTE_AUTHID 'root', ADD  REMOTE_PASSWORD 'elvis'
);
```

The Wrapper Manager module performs the second task by producing the following SQL statements:

```
CREATE WRAPPER "odbc_sources" LIBRARY 'db2rcodbc.dll'
OPTIONS
(
    ADD DB2_FENCED 'N'
);

CREATE SERVER "odbc_server" TYPE ODBC VERSION '3.0' WRAPPER "odbc_sources"
OPTIONS
(
    ADD NODE 'mysql_db'
);
```

Note that `mysql_db` is the system data source name defined to access the MySQL database, and contains the references to the specific MySQL database. Once the statements for the instantiation of the wrapper and the server have been created, the module produces the statement that generates the nickname that realizes the connection with the MySQL table:

```
CREATE NICKNAME s3 FOR mysql_db."players";
```

```
CREATE VIEW player(name, team, age) AS
SELECT
 S.name, S.team, S.age
FROM
 s3 S;
```

Figure 5.4: View definition for the player global relation of Example 5.2.3

Note that `players` is the MySQL table we are wrapping into the data integration system. ∎

5.2.4 Mapping Processor

The Mapping Processor module reads the mapping specification provided by the Specification Processor and produce an SQL script containing a set of view creation statements. As mentioned earlier, the system is able to handle mappings expressed by means of the GAV formalism, in which each global relation is defined as a view over the relations of the source schema. Hence, the task accomplished by the Mapping Processor, consists in defining a view for every GAV mapping assertion, i.e., for every global relation. This set of views will represent the mediated layer of the data integration system and will be the schema accessible by the user. The following example shows how the Mapping Processor module works.

Example 5.2.3 Let us consider again the system specification presented in Example 5.2.1: such specification contains two global relations (player and team) suitably wrapped to the source schema. For each global relation the Mapping Processor produces a view definition that uses the nicknames defined by the Source Processor. The resulting view definitions for the player and team global relations are presented respectively in Figure 5.4 and Figure 5.5. ∎

5.2.5 Query Reformulator

Query Reformulator is the module responsible for suitably transforming the user queries in order to allow DB2II to evaluate it. The reformulated queries are then passed to DB2II, which is in charge of evaluating it over the data integration system specification $\mathcal{I}'$ produced at setup mode. The results of such evaluation are then sent back to the user (see Figure 5.1).

As we will explain better in the following, the module implements two alternative techniques for query rewriting. We refer the reader to Section 5.3 for further details on the query processing algorithm implemented in the Query Reformulator module.

5.3 Data integration techniques

In this section, we define two different techniques that can be adopted by the system for integrating data, and discuss the correctness of both the approaches.

The solutions we propose have the aim to implement efficient query processing algorithms that exploits all the optimization techniques supplied by the commercial tools presented earlier.

```
CREATE VIEW team(name, leader) AS
(
 SELECT
   S1.name, S1.leader
 FROM
   s1 S
 UNION
 SELECT
   S2.name, S2.leader
 FROM
   s2 S
);
```

Figure 5.5: View definition for the team global relation of Example 5.2.3

Note that, in the specification of the logical framework for data integration, we have assumed to deal with a relational source schema $\mathcal{S}$. There, we implicitly assumed to take advantage from a layer of wrappers that hide the real nature of the data sources, as that provided by DB2II.

Informally, the two techniques differ one another for the workload assigned to the data federation tool in the setup mode. Indeed, in the first technique, called *internal technique*, for answering a user query, we rely on the ability of DB2II in exploiting view definitions for query evaluation, whereas in the second technique (called *external*) we simply rely on its ability in evaluating queries specified over relational tables. Notice that in our system we pay the fact that DB2II is heavily exploited at run mode with the external technique by performing a more complex setup mode with the internal technique.

More precisely, the two techniques proceed as follows:

- **Internal technique:** the idea is to rely upon the management of views provided by DB2II. The compiled DB2II specification contains a view V_R, for each global relation R which corresponds to the query that maps R to the source schema. In this approach, the effort of the Query Reformulator module is minimal.

- **External technique:** the idea is to focus the process on the user queries, rather than on the system specification, implementing an *unfolding* technique [92] that takes into account the correspondences expressed in the mappings. In this case the DB2II specification does not need to contain views.

In the next two sections we will describe both the techniques.

5.3.1 Internal technique: implementing data integration using views

Since DB2 Information Integrator relies on DB2 Universal Database, a possible way to implement query processing is to take advantage of the management of views of DB2II, in order to implement the correspondences expressed in the mapping. In particular, the DB2II compiled instance $\mathcal{I}'$

Algorithm $compile_i(\mathcal{I})$
Input:
A data integration systems specification $\mathcal{I}$
begin
$\mathcal{V} := \{\}$;
foreach $R(\mathbf{A}) \in \mathcal{G}$ **do**
$\mathcal{V} := \mathcal{V} \cup \langle V_R(\mathbf{A}), V_R \leftarrow body(\rho(R))\rangle$
return $\mathcal{V}$
end

Figure 5.6: $compile_i(\mathcal{I})$ algorithm.

that results from the compilation of an input data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M}\rangle$ can be represented by the couple $\mathcal{I}' = \langle \mathcal{V}, \mathcal{S}'\rangle$ such that:

- $\mathcal{S}'$ is the set of nicknames obtained from the source schema of $\mathcal{I}$ by the execution of the Source Processor;
- $\mathcal{V}$ is a a set of views produced by the Mapping Manager. Each view is represented as a couple $\langle V(\mathbf{A}), \Psi(V)\rangle$ such that:
  - $V(\mathbf{A})$ is the view schema, where $\mathbf{A}$ is a sequence of attributes of the view and V is the view name;
  - $\Psi(V)$ is the conjunctive query on $\mathcal{S}$ that defines the view; $\Psi(V)$ has the form:
$$V(\vec{x}) \leftarrow S_1(\vec{x_1}, \vec{y_1}), \ldots, S_k(\vec{x_k}, \vec{y_k})$$

In practice, the system implements a compilation algorithm $compile_i(\mathcal{I})$ (the subscript i means that the algorithm refers to the internal technique), shown in Figure 5.6, that generates the set $\mathcal{V}$ of views of $\mathcal{I}_i{}'$, by applying the following rule: if $R(\mathbf{A}) \in \mathcal{G}$, then insert into $\mathcal{V}$ the couple $\langle V_R(\mathbf{A}), \Psi(V_R)\rangle$, where:

- $\rho(R) = R(\vec{x}) \leftarrow r_1(\vec{x_1}, \vec{y_1}), \ldots, r_k(\vec{x_k}, \vec{y_k})$. In words, $\rho(R)$ returns the mapping view associated to the global relation R.
- $\Psi(V_R) = V_R(\vec{x}) \leftarrow r_1(\vec{x_1}, \vec{y_1}), \ldots, r_k(\vec{x_k}, \vec{y_k})$.

Together with the compilation algorithm the system also provides a translation mechanism that, given a query q over $\mathcal{I}$, returns a corresponding query q' over the set of views $\mathcal{V}$ that belong to $\mathcal{I}'$. Such a translation algorithm $translate_i(Q, \mathcal{I})$, shown in Figure 5.7, generates the result, by applying the two following syntactical rules:

1. the head of the query q' is identical to the head of the query q;
2. for each atom $R(\vec{x_1}, \vec{y_1})$ in the body of q, insert the atom $V_R(\vec{x_1}, \vec{y_1})$ in the body of q'.

Algorithm $translate_i(Q, \mathcal{I})$
Input:
A data integration systems specification $\mathcal{I}$
A union of conjunctive queries Q
begin
$q_{aux} := \{\}$;
foreach $g \in body(Q)$ **such that** $g = R(\vec{x_i}, \vec{y_i})$ **do**
$q_{aux} := q_{aux} \cup V_R(\vec{x_i}, \vec{y_i})$
$Q' := head(Q) \leftarrow q_{aux}$
return Q' **end**

Figure 5.7: The $translate_i(Q, \mathcal{I})$ algorithm.

Example 5.3.1 Let us consider a data integration system specification $\mathcal{I}$ containing three data sources: the first one stores information coming from the Registry Office concerning citizens and enterprises, the second one holds geographical information about the cities and their dislocation, the third one stores information coming from the Land Register about ownerships (buildings and lands). The source schema $\mathcal{S}$ that represents these sources within the system contains four relations:

$$\begin{array}{lcl} s_1 & = & \mathsf{citizen}(\text{ssn, name, citycode}) \\ s_2 & = & \mathsf{enterprise}(\text{ssn, name, citycode, emp\_number}) \\ s_3 & = & \mathsf{city}(\text{code, name, country}) \\ s_4 & = & \mathsf{ownerships}(\text{code, owner\_ssn, address, type}) \end{array}$$

We want to extract from these sources, only information about owners and their buildings. We can define the global schema $\mathcal{G}$ with two relations

$$\begin{array}{lcl} R_1 & = & \mathsf{owner}(\text{name, city}) \\ R_2 & = & \mathsf{building}(\text{address, owner}) \end{array}$$

and the mapping $\mathcal{M}$ can be defined as follows

$$\begin{array}{lcl} v_1 & = & \mathsf{owner}(X, Y) \leftarrow \mathsf{citizen}(W_1, X, Z), \mathsf{city}(Z, Y, W_2). \\ v_2 & = & \mathsf{owner}(X, Y) \leftarrow \mathsf{enterprise}(W_1, X, Z, W_2), \mathsf{city}(Z, Y, W_3). \\ v_3 & = & \mathsf{building}(X, Y) \leftarrow \mathsf{ownership}(W_1, Y, X, 'building'). \end{array}$$

Consider now a query q that asks for owners' names and addresses of buildings of Rome:

$$q \quad = \quad \mathsf{q}(X, Y) \leftarrow \mathsf{owner}(X, 'Rome'), \mathsf{building}(Y, X).$$

The compiled DB2II instance $\mathcal{I}'$ obtained from $\mathcal{I}$ is:

$$\begin{array}{rcrcl} \langle V_{\mathsf{owner}}(name, city) & , & \{V_{\mathsf{owner}}(X, Y) & \leftarrow & \mathsf{citizen}(W_1, X, Z), \mathsf{city}(Z, Y, W_2)., \\ & & V_{\mathsf{owner}}(X, Y) & \leftarrow & \mathsf{enterprise}(W_1, X, Z, W_2), \\ & & & & \mathsf{city}(Z, Y, W_3).\}\rangle \\ \langle V_{\mathsf{building}}(address, owner) & , & V_{\mathsf{building}}(X, Y) & \leftarrow & \mathsf{ownership}(W_1, Y, X, 'building').\rangle \end{array}$$

Note that the view definition for V_{owner} is a union of conjunctive queries. The translation for the user query q to be posed on the defined views is

$$q \quad = \quad \mathsf{q}(X,Y) \leftarrow V_{\mathsf{owner}}(X,'Rome'), V_{\mathsf{building}}(Y,X).$$

■

5.3.2 External technique: external management of global schema and mapping

In the second solution, we specify a technique for the implementation of a data integration system specification by means of external data structure. In short, we leave inside the DB2II compiled instance $\mathcal{I}'$ only the source schema $\mathcal{S}$, that is, the set of nicknames that wrap the sources, maintaining the global schema and the mapping between global relations and nicknames outside DB2II (by means of appropriated external software components).

Then, since the user poses poses his queries on the global schema we have to preliminarily transform such queries into queries expressed in terms of the source schema. Informally, this may be done by substituting each atom appearing in the body of the query with the body of its corresponding mapping definition. This process can be seen as an extension of the well-known *unfolding* algorithm [92], and can be formalized as follows. Given a data integration system specification $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ and a CQ q over $\mathcal{I}$, we call *translated query* for $\mathcal{I}'$ the query $q' = translate_e(q, \mathcal{I})$ (the subscript e means that the algorithm refers to the external technique) where $translate_e(q, \mathcal{I})$ is the unfolding algorithm, shown in Figure 5.8, that we adopt for query reformulation.

Such an algorithm makes use of the subroutines: `mapping_by_head(`g`)`, that given an atom g returns the mapping view whose head refers to g; `unify(`g_1`,`g_2`)`, that unifies the variables of two atoms g_1 and g_2, and returns the most general unifier (m.g.u.) between g_1 and g_2; `replace(`q`,`g`,`$conj$`)`, that, given a conjunctive query q, one of its body atoms g, and a conjunction of atoms $conj$, returns a query obtained by replacing the atom g of the body of the query q with the conjunction $conj$.

Example 5.3.2 We consider again the system specification $\mathcal{I}$ presented in Example 5.3.1. As we have said before, there is no need of compiling the specification, but we have to unfold the query q over the source relations. Based on the unfolding algorithm, we obtain the following union of conjunctive query

$$\begin{array}{lcl} q'(X,Y) & \leftarrow & \mathsf{citizen}(W_1,X,Z), \mathsf{city}(Z,Y,W_2), \\ & & \mathsf{ownership}(W_3,Y,X,'building'). \\ q'(X,Y) & \leftarrow & \mathsf{enterprise}(W_1,X,Z,W_2), \mathsf{city}(Z,Y,W_3), \\ & & \mathsf{ownership}(W_4,Y,X,'building'). \end{array}$$

■

Algorithm $translate_e(q, \mathcal{I})$
Input:
A data integration systems specification $\mathcal{I}$
A union of conjunctive queries Q
begin
$q_{aux} := q$;
foreach global atom g in $body(q)$
$v :=$ mapping_by_head(g);
$\sigma :=$ unify(g, $head(v)$);
if($\sigma == \{\}$)
return NULL;
else
$q_{aux} := \sigma[\text{replace}(q, g, body(v))]$;
end if
return q_{aux};
end

Figure 5.8: $translate_e(q, \mathcal{I})$ algorithm

5.3.3 Correctness

So far we have proposed two different techniques that allow the implementation of a data integration system in DB2II. Compilation and translation algorithms basically consist, in both the techniques, of syntactical transformations.

The following theorems show the correctness of both the solutions. The proofs straightforwardly follow from basic logic programming notions [77]. Note that in the theorems we will indicate with $\mathcal{I}_i{}'$ and $q_i{}'$ the specification and the query obtained by means of the algorithm $compile_i$ and $translate_i$ of the internal technique. Conversely, with $\mathcal{I}_e{}'$ and $q_e{}'$, the specification and the query obtained by means of the algorithm $compile_e$ and $translate_e$ referring to the external technique.

Theorem 5.3.3 *Given a data integration system specification $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, a database instance $\mathcal{D}$ for $\mathcal{I}$, a conjunctive query q over $\mathcal{G}$ and a tuple $\bar{t}$, $\bar{t} \in q^{\mathcal{I},\mathcal{D}}$ if and only if $\bar{t}$ belongs to the set of answers we obtain by querying the compiled DB2II instance $\mathcal{I}_i{}'$ by means of the query $q_i{}'$, where $\mathcal{I}_i{}'$ is the DB2II instance obtained from the compilation of $\mathcal{I}$ by means of the algorithm $compile_i(\mathcal{I})$, and $q_i{}'$ is the query obtained from the translation obtained of q translation by means of the algorithm $translate_i(q, \mathcal{I})$.*

Theorem 5.3.4 *Given a data integration system specification $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, a database instance $\mathcal{D}$ for $\mathcal{I}$, a conjunctive query q over $\mathcal{G}$ and a tuple $\bar{t}$, $\bar{t} \in q^{\mathcal{I},\mathcal{D}}$ if and only if $\bar{t}$ belongs to the set of answers we obtain by querying the DB2II instance $\mathcal{I}_e{}'$, where $\mathcal{I}_e{}'$ is constituted only by $\mathcal{S}$, and $q_e{}'$ is the query obtained from the translation obtained from q by means of the $translate_e(q, \mathcal{I})$ algorithm.*

Chapter 6

Efficient Query Answering for Data Integration Systems

In Chapter 5 we presented the architecture of a novel system based on commercial tools that enables the designer to integrate a number of heterogeneous and independent information sources under a common virtual schema accessible by the users. The connection with the sources is obtained by the wrapping features provided by the underlying commercial system, and by defining an expressive mapping between the virtual global schema and the source schema constituting the internal representation of the wrapped sources.

Our system provides a significant improvement with respect to the features offered by standard commercial tools: indeed, the capability of handling global schema and mapping is actually missing in all the commercial systems analyzed in this thesis. However, in order to meet the requirements of the today's information integration scenario, also the ability to efficiently handle expressive integration environments, in which the designer can express integrity constraints (ICs) over the global schema, is needed. To this aim one would like to somehow incorporate in the system the state-of-the-research techniques for handling the presence of integrity constraints and of possible data inconsistency (i.e., consistent query answering algorithms).

Unfortunately, most of the theoretical solutions coming from the research field to such kind of problem, are hardly applicable to practical real cases due to the inherent complexity of the problem.

In this chapter we face the problem of *efficiently* querying data integration systems in the presence of integrity constraints expressed over the global schema. With the ultimate goal of exploiting current DBMSs for data integration in this setting. We first study the problem from the theoretical viewpoint in a simplified scenario with a single possibly inconsistent database, and then extend the solution to the more complex setting of a data integration system.

The chapter is structured as follow: in Section 6.1 we present some limitations of the semantics introduced in Chapter 2, and propose a new semantics for dealing with integrity constraints in data integration systems; then, we recall the basic notions of *Consistent Query Answering* (CQA) in Section 6.2; in Section 6.3 we provide efficient algorithms for consistent query answering in

the presence of several kinds of integrity constraints and for different query languages; finally, in Section 6.4, we explain how the algorithms introduced can be exploited in a data integration environment.

6.1 Limits of standard semantics for Data Integration Systems

As we have already mentioned, data sources are autonomous and independent: it follows that in general data retrieved from sources cannot be reconciled in a global schema $\mathcal{G}$ in such a way that both the assumption on the mapping and the integrity constraints on $\mathcal{G}$ are satisfied. Hence, the situation may arise in which no global databases exist in the semantics of the data integration system. Notice that this happens even if most of the data satisfy global constraints and there is a single violation of a single integrity constraint. In these cases the data integration system is considered *inconsistent*. The following example shows that also considering very simple schemas and mappings, the presence of integrity constraints may lead to the non interesting case of empty semantics.

Example 6.1.1 Let us consider a data integration system $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ in which

- $\mathcal{G} = \langle \mathcal{R}_\mathcal{S}, \mathcal{K} \rangle$ is the global schema in which $\mathcal{R}_\mathcal{S}$ contains two binary relations, $r/2$ and $s/2$, and $\mathcal{K}$ contains the following integrity constraints: $key(r) = \{1\}$ and $key(s) = \{1\}$;
- $\mathcal{S} = \{s_1/2, s_2/2\}$ is the source schema;
- $\mathcal{M}$ is a GAV mapping containing the following assertions:

$$\begin{array}{lcl} r_1(X,Y) & :- & s_1(X,Y) \\ r_2(X,Y) & :- & s_2(X,Y). \end{array}$$

Consider now the following source database $\mathcal{DB} = \{\mathsf{r(a,b)}, \mathsf{r(a,c)}\}$: under the complete semantics, we have the following global database for $\mathcal{I}$ with respect to $\mathcal{DB}$:

$$\begin{array}{lcl} \mathcal{B}_1 & = & \{\mathsf{r(a,b)}\} \\ \mathcal{B}_2 & = & \{\mathsf{r(a,c)}\} \\ \mathcal{B}_3 & = & \emptyset \end{array}$$

Anyway, for any query q over $\mathcal{I}$, we have that $ans_c(q, \mathcal{I}, \mathcal{DB}) = \emptyset$ since $\emptyset \in sem_c(\mathcal{I}, \mathcal{DB})$. Furthermore, with respect to the sound and exact semantics, we have that no global database exists that satisfies both the mapping and the integrity constraints of $\mathcal{I}$: it follows that $sem_e(\mathcal{I}, \mathcal{D}) = sem_s(\mathcal{ID}) = \emptyset$. ∎

In the above example query answering under the exact or sound semantics is meaningless in the presence of inconsistency. Indeed, according to the "ex falso quod libet" principle, every tuple is in the answer to every query. However, one would like to obtain significant answers to queries even in the presence of inconsistencies. The main point is to establish what are the

answers that have to be returned to queries posed to an inconsistent data integration system. The standard approach to this problem is through data cleaning [13, 87], i.e., by explicitly modifying the data in order to eliminate violation of ICs: only when data are "repaired", i.e., are consistent with the ICs, queries can be answered. However, in many situations it would be much more desirable to derive significant information even in the presence of inconsistent data. The interest in this research area is motivated by the fact that, in data integration as well as in many other application scenarios, the explicit repair of data is not convenient, or even not possible.

A possible way to deal with data integration systems that are inconsistent with respect to a set of integrity constraints expressed over the global schema (as for Example 6.1.1), is to modify the semantics of the system to make it inconsistency-tolerant. In the following we will present such a semantics in the simplified scenario of a single inconsistent database. Then we extend such semantics to the more general settings of data integration system described in Chapter 2.

6.2 Consistent Query Answering

We now study the problem of Consistent Query Answering (CQA). We consider here a simplified scenario with a single database, instead of the more complex scenario of a data integration system. Notice that this single database can be viewed as the global schema of a data integration system in which the mapping is constituted by simple one-to-one correspondences between global and source relations (see for example the data integration system presented in Example 6.1.1).

Consistent query answering studies the definition (and computation) of "meaningful" answers to queries posed to databases whose data do not satisfy the ICs declared on the database schema [7, 51, 19]. All these approaches are based on the following principle: *schema is stronger than data*. In other words, the database schema (i.e., the set of integrity constraints) is considered as the actually reliable information (strong knowledge), while data are considered as information to be revised (weak knowledge). Therefore, the problem amounts to deciding how to "repair" (i.e., change) data in order to reconcile them with the information expressed in the schema. Therefore, the intuitive semantics of consistent query answering can be expressed as follows: a tuple t is a consistent answer to a query q in an inconsistent database $\mathcal{D}$ if t is an answer to q in all the *repairs* of $\mathcal{D}$, i.e., in all the possible databases obtained by (minimally) modifying the data in $\mathcal{D}$ to eliminate violations of ICs. The non-monotonic character of consistent query answering can be immediately seen through the following simple example.

Example 6.2.1 Let $D = \{r(a, b)\}$ be a database whose schema contains the declaration of a key dependency on the first attribute of r. Since the database instance does not violate the key dependency on r, the only repair of the database is D itself. Hence, the following query $q(X, Y) \leftarrow r(X, Y)$ has the consistent answer $t = \langle a, b\rangle$. Now, let D' be the database instance obtained by adding the fact $r(a, c)$ to D. D' is inconsistent with the key dependency, and has two possible repairs: $\{r(a, b)\}$ and $\{r(a, c)\}$. Since there is no tuple which is an answer to q in both repairs, it follows that there are no consistent answers to the query q in D'. In contrast, observe that the query $q'(X) \leftarrow r(X, Y)$ has the answer $\langle a\rangle$ both in D and in D', which can be therefore considered consistent. ∎

Recent studies in this area have established declarative semantic characterizations of consistent query answering over relational databases, decidability and complexity results for consistent query answering, as well as techniques for query processing [7, 35, 51, 19, 14, 20]. In particular, it has been shown that computing consistent answers of conjunctive queries (CQs) is coNP-hard in data complexity, i.e., in the size of the database instance, even in the presence of very restricted forms of ICs (single, unary keys).

From the algorithmic viewpoint, the approach mainly followed is query answering via query rewriting: (i) First, the query that must be processed (usually a conjunctive query) is reformulated in terms of another, more complex query. Such a reformulation is purely intensional, i.e., the rewritten query is independent of the database instance; (ii) Then, the reformulated query is evaluated over the database instance. Due to the semantic nature and the inherent complexity of consistent query answering, Answer Set Programming (ASP) is usually adopted in the above reformulation step [51, 14, 20], and stable model engines like Smodels [81], and DLV [72] can be used for query processing.

An orthogonal approach to consistent query answering is the one followed by recent theoretical works [7, 35, 47], whose aim is to identify *subclasses* of CQs whose consistent answers can be obtained by rewriting the query in terms of a *first-order* (FOL) query. The advantage of such an approach is twofold: first, this technique allows for computing consistent answers in time polynomial in data complexity (i.e., for such subclasses of queries, consistent query answering is computationally simpler than for the whole class of CQs); second, consistent query answering in these cases can be performed through standard database technology, since the FOL query synthesized can be easily translated into SQL and then evaluated by any relational DBMS. On the other hand, this approach is only limited to polynomial subclasses of the problem. In particular, Fuxman and Miller in [47] have studied databases with key dependencies, and have identified a broad subclass of CQs that can be treated according to the above strategy.

In the next sections we study several classes of queries and integrity constraints for which consistent query answering can be performed efficiently, i.e., in polynomial time in data complexity. In particular we analyze (union of) conjunctive queries in the presence of key dependencies, exclusion dependencies and inclusion dependencies, a well-known class of ICs. In particular the ability to deal with KDs and IDs ensure the possibility of dealing with foreign keys, undoubtly the most important integrity constraints used in database management systems. Moreover, exclusion dependencies are not only typical of relational database schemas, but are also relevant and very common in languages for conceptual modelling, e.g., ontology languages [8]: indeed such dependencies allow for modelling partitioning/disjointness of entities. This makes the study of this set of integrity constraints particularly important for the broad applicability of consistent query answering.

6.3 Efficient Algorithms for CQA

As already mentioned in previous sections, computing the consistent answers to queries posed over inconsistent databases is a very hard task also in the presence of very restricted forms of integrity constraints. Some recent works in that area [47, 46] aim to identifying classes of queries

for which the problem is computationally easier. The approach followed by the authors of those works, consists in rewriting the original query into a new first-order logic query, whose evaluation over the inconsistent database returns only the consistent answers with respect to the integrity constraints posed over the database schema.

In this section we study the problem of solving CQA via first-order rewriting of queries in the presence of several kinds of integrity constraints. We first state the formal framework for CQA in Section 6.3.1, and then study key dependencies for conjunctive queries and union of conjunctive queries in Section 6.3.2, Section 6.3.3 and Section 6.3.4 respectively. In Section 6.3.5 we attack the problem of CQA under key dependencies and inclusion dependencies and, finally, in Section 6.3.6 we propose an algorithm to deal with both key dependencies and exclusion dependencies for single conjunctive queries.

6.3.1 Formal framework for CQA

Syntax

We consider to have an infinite, fixed alphabet Γ of constants representing real world objects, and we take into account only database instances having Γ as domain. Moreover, we assume that different constants in Γ denote different objects, i.e., we adopt the so-called *unique name assumption*.

A *database schema* $\mathcal{S}$ is constituted by a a relational signature $\mathcal{A}$, and a set of integrity constraints $\mathcal{K} \cup \mathcal{E} \cup \mathcal{F}$ over the symbols of $\mathcal{A}$ (see Section 2.1.1).

A *term* is either a variable or a constant of Γ. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$ where p is a relation symbol of arity n and $t_1, \ldots, t_n$ is a sequence of n terms. An atom is called *fact* if all the terms occurring in it are constants. A *database instance* $\mathcal{D}$ for $\mathcal{S}$ is a set of facts over $\mathcal{A}$. We denote as $r^{\mathcal{D}}$ the set $\{\vec{\mathbf{t}} \mid r(\vec{\mathbf{t}}) \in \mathcal{D}\}$.

A *union of conjunctive queries* (UCQ) q of arity n over a (database schema with) signature $\mathcal{A}$ is an expression of the form $h(x_1, \ldots, x_n) \coloneq d_1 \vee \ldots \vee d_m$, where the atom $h(x_1, \ldots, x_n)$ is called the *head* of the query (denoted by *head*(q)), $d_1 \vee \ldots \vee d_m$ is called the *body* of the query (denoted by *body*(q)), and for each $i \in \{1 \ldots m\}$, d_i, called the *i-th disjunct* of q, is a conjunction of atoms $a_{i,1} \wedge \ldots \wedge a_{i,k}$, whose predicate symbols are in $\mathcal{A}$, such that all the variables occurring in the query head also occur in d_i. If $m = 1$, q is simply called *conjunctive query* (CQ). In a UCQ q, we say that a variable is a *head variable* if it occurs in the query head, while we say that a variable is *existential* if it only occurs in the query body. Moreover, we call an existential variable *shared in a disjunct d of q* if it occurs at least twice in d (otherwise we say that it is *non-shared in d*). Obviously, if q is a CQ, an existential variable shared (resp. non-shared) in the unique disjunct of q will be simply called shared (resp. non-shared) in q.

A *FOL query* of arity n is an expression of the form $\{x_1, \ldots, x_n \mid \Phi(x_1, \ldots, x_n)\}$, where $x_1, \ldots, x_n$ are variable symbols and Φ is a first-order formula with free variables $x_1, \ldots, x_n$.

Semantics

First, we briefly recall the standard evaluation of queries over a database instance. Let q be the UCQ $h(x_1, \ldots, x_n) \coloneq d_1, \ldots, d_m$ and let $\vec{\mathbf{t}} = \langle c_1, \ldots, c_n \rangle$ be a tuple of constants of Γ. A set

of facts I is an *image of* $\vec{\mathbf{t}}$ *w.r.t.* q if there exists a substitution σ of the variables occurring in a disjunct d_i of q such that $\sigma(head(q)) = h(\vec{\mathbf{t}})$ and $\sigma(d_i) = I$. Given a database instance $\mathcal{D}$, we denote by $q^{\mathcal{D}}$ the evaluation of q over $\mathcal{D}$, i.e., $q^{\mathcal{D}}$ is the set of tuples $\vec{\mathbf{t}}$ such that there exists an image I of $\vec{\mathbf{t}}$ w.r.t. q such that $I \subseteq \mathcal{D}$.

Given a FOL query q and a database instance $\mathcal{D}$, we denote by $q^{\mathcal{D}}$ the evaluation of q over $\mathcal{D}$, i.e., $q^{\mathcal{D}} = \{\langle c_1, \ldots, c_n\rangle \mid \mathcal{D} \models \Phi(c_1, \ldots, c_n)\}$, where each t_i is a constant symbol and $\Phi(c_1, \ldots, c_n)$ is the first-order sentence obtained from Φ by replacing each free variable x_i with the constant c_i.

Then, we define the *loosely sound* semantics of inconsistent databases. We first recall when a database instance satisfies the ICs expressed over its schema:

- A database instance $\mathcal{D}$ *violates* the KD $key(r) = \{i_1, \ldots, i_k\}$ if there exist two *distinct* facts $r(c_1, \ldots, c_n)$, $r(d_1, \ldots, d_n)$ in $\mathcal{D}$ such that $c_{i_j} = d_{i_j}$ for each j such that $1 \leq j \leq k$.
- A database instance $\mathcal{D}$ *violates* the ED $r[\mathbf{A}] \cap s[\mathbf{B}] = \emptyset$ if there exist two tuples t_1 and t_2 belonging to the extension of r and s respectively, such that $t_1[\mathbf{A}] = t_2[\mathbf{B}]$, where $t_1[\mathbf{A}]$ (resp. $t_2[\mathbf{B}]$) is the projection over the attributes $\mathbf{A}$ of the tuple t_1 (resp. t_2) of r (resp. of s).
- A database instance $\mathcal{D}$ *violates* the ID $r[\mathbf{A}] \subseteq s[\mathbf{B}]$ if there exist a tuple t_2 of s for which a tuple t_1 of r there not exist such that $t_1[\mathbf{A}] = t_2[\mathbf{B}]$.

It follows that, a database instance $\mathcal{D}$ is *legal for* $\mathcal{S}$ if $\mathcal{D}$ does not violate any integrity constraints in $\mathcal{K}, \mathcal{E}$ and $\mathcal{F}$.

A set of ground atoms $\mathcal{D}'$ is a *repair of* $\mathcal{D}$ *under* $\mathcal{S}$ iff:

(i) $\mathcal{D}'$ is legal for $\mathcal{S}$;

(ii) for each $\mathcal{D}''$ such that $\mathcal{D}''$ is legal for $\mathcal{S}$, $\mathcal{D}'' \cap \mathcal{D} \not\supset \mathcal{D}' \cap \mathcal{D}$.

In words, a repair for $\mathcal{D}$ under $\mathcal{S}$ is a maximal subset of $\mathcal{D}$ that is legal for $\mathcal{S}$.

The problem which we are interested in is the following: given a database schema $\mathcal{S}$, a database instance $\mathcal{D}$, and a UCQ q, return all tuples $\vec{\mathbf{t}}$ of constants of Γ such that, for each repair $\mathcal{D}'$ of $\mathcal{D}$ under $\mathcal{S}$, $\vec{\mathbf{t}} \in q^{\mathcal{D}'}$. Each such tuple is called *consistent answer* to q in $\mathcal{D}$ under $\mathcal{S}$. We denote by $ConsAns(q, \mathcal{S}, \mathcal{D})$ the set of consistent answers to q in $\mathcal{D}$ under $\mathcal{S}$. Notice that the loosely-sound semantics is *not* the most commonly adopted one in CQA: in general, works on CQA refers to the *loosely-exact* semantics, that is, a semantics equal to the loosely-sound except for the point (ii) above, that becomes:

(ii) for each $\mathcal{D}''$ such that $\mathcal{D}''$ is legal for $\mathcal{S}$, $\mathcal{D}''\Delta\mathcal{D} \not\subset \mathcal{D}'\Delta\mathcal{D}$.

where Δ denote the symmetric difference between sets, i.e., given two sets A and B, $A\Delta B = (A - B) \cup (B - A)$. Roughly speaking, the loosely-exact semantics allows for tuple deletion and addition, in order to "repair" an inconsistent database instance. It is well known that tuple deletion is the only possible way to repair denial constraints (KDs and EDs, in our settings), whereas an inclusion dependencies can be repaired by deleting or inserting tuples. Conversely,

in the loosely-sound semantics tuple insertion is the preferred way to repair inconsistencies. The following example will clarify the matter.

Example 6.3.1 Let us consider a database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{F} \rangle$ in which $\mathcal{A}$ contains two relations $r/2$ and $s/2$, $\mathcal{K}$ contains the key dependencies $key(r) = \{1\}$ and $key(s) = \{1\}$, and $\mathcal{F}$ contains the inclusion dependency $r[2] \subseteq s[1]$. Let us consider now the following database instance:

$$\mathcal{D} = \{\mathsf{r(a, b)}, \mathsf{r(a, c)}\}$$

Based on the loosely-exact semantics, possible repairs are for example the sets $\mathcal{D}_1 = \{r(a, b)\}$ or $\mathcal{D}_2 = \{r(a, c)\}$, in which a tuple has been deleted in order to repair the key constraint violation. Anyway, also the following set $\mathcal{D}_3$ is a possible repair of $\mathcal{D}$ based on the loosely-exact semantics:

$$\mathcal{D}_3 = \{\mathsf{r(a, b)}, \mathsf{s(b, }\alpha\mathsf{)}\}$$

in which the tuple $\mathsf{r(a,c)}$ has been removed in order to repair the key dependency on r, and the tuple $\mathsf{s(b,}\alpha\mathsf{)}$, where α is a fresh constant of Γ, has been added in order to repair the inclusion dependency $r[2] \subseteq s[1]$. The main difference between the loosely-exact and the loosely-sound semantics is that in the second one, repair $\mathcal{D}_3$ is "preferred" with respect to repairs $\mathcal{D}_1$ and $\mathcal{D}_2$.
■

Anyway, an important correspondence between the two semantics above defined is the following: the consistent answers obtained with the loosely-sound and loosely-exact semantics coincide in all cases of CQA in which only denial constraints are considered.

Analogously to [47], we say that consistent query answering for a class $\mathcal{C}$ of UCQs is *FOL-reducible* (or simply that the class $\mathcal{C}$ is FOL-reducible), if for every database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$ and every query $q \in \mathcal{C}$ over $\mathcal{A}$, there exists a FOL query q_f over $\mathcal{A}$ such that for every database instance $\mathcal{D}$, $\vec{\mathbf{t}}$ is a consistent answer to q in $\mathcal{D}$ under $\mathcal{S}$ iff $\vec{\mathbf{t}} \in q_f^{\mathcal{D}}$. We call such a q_f a *FOL-rewriting* of q under $\mathcal{S}$. Notice that FOL-reducibility is a very interesting property from a practical point of view, since FOL queries correspond to queries expressed in relational algebra (i.e., in SQL). Observe also that every FOL query can be evaluated in LogSpace wrt data complexity, i.e., computational complexity w.r.t. the size of the database instance (see e.g., [5]). It follows that if a class $\mathcal{C}$ is FOL-reducible, then consistent query answering for $\mathcal{C}$ is in LogSpace wrt data complexity.

Example 6.3.2 Consider the database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$, where $\mathcal{A}$ comprises the relations $\mathsf{Journal}(\mathit{title}, \mathit{editor})$, $\mathsf{ConfPr}(\mathit{title}, \mathit{editor})$, $\mathsf{Editor}(\mathit{name}, \mathit{country})$, and the set of integrity constraints $\mathcal{K}$ comprises the key dependencies $key(\mathit{Journal}) = \{1\}$, $key(\mathit{ConfPr}) = \{1\}$, $key(\mathit{Editor}) = \{1\}$. Consider the database instance $\mathcal{D}$ described below

$$\begin{array}{c}\mathcal{D} = \{\mathsf{Journal(TODS, ACM)}, \mathsf{Journal(TODS, IEEE)},\\ \mathsf{Editor(ACM, USA)}, \mathsf{ConfPr(PODS05, ACM)},\\ \mathsf{ConfPr(PODS05, SV)}, \mathsf{Editor(IEEE, USA)}\}.\end{array}$$

It is easy to see that $\mathcal{D}$ is not consistent with the KDs on Journal and ConfPr of $\mathcal{S}$. Then, the repairs of $\mathcal{D}$ under $\mathcal{S}$ are:

$$\mathcal{D}_1 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{ACM}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{ACM}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{USA})\}$$

$$\mathcal{D}_2 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{ACM}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{SV}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{USA})\}$$

$$\mathcal{D}_3 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{IEEE}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{ACM}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{USA})\}$$

$$\mathcal{D}_4 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{IEEE}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{SV}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{USA})\}.$$

Let $q(x, z) :\!- \mathit{Journal}(x, y), \mathit{Editor}(y, z)$ be a user query. The consistent answers to q in $\mathcal{D}$ under $\mathcal{S}$ are $\{\langle \mathsf{TODS}, \mathsf{USA} \rangle\}$. ∎

6.3.2 First-order reducibility of CQs under KDs

It is well known that the consistent query answering problem studied in this chapter is coNP-hard in data complexity for generic conjunctive queries [19, 35] in the presence of key dependencies. The example below show the exponential blowup of repairs that causes the high complexity of CQA.

Example 6.3.3 Let us consider again the database instance $\mathcal{D}$ of Example 6.3.2. Note that the presence of two duplicated keys in the relation Journal and ConfPr produces four possible repairs ($\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ and $\mathcal{D}_4$). Consider now the database instance $\mathcal{D}'$ obtained from $\mathcal{D}$ by adding the inconsistent tuple $\mathsf{Editor}(\mathsf{IEEE}, \mathsf{EU})$. It should be easy to see that now the number of possible repairs doubles to eight. Indeed, together with the repairs $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ and $\mathcal{D}_4$ we now have also the following:

$$\mathcal{D}_5 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{ACM}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{ACM}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{EU})\}$$

$$\mathcal{D}_6 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{ACM}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{SV}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{EU})\}$$

$$\mathcal{D}_7 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{IEEE}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{ACM}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{EU})\}$$

$$\mathcal{D}_8 = \{\mathsf{Journal}(\mathsf{TODS}, \mathsf{IEEE}), \mathsf{ConfPr}(\mathsf{PODS05}, \mathsf{SV}), \mathsf{Editor}(\mathsf{ACM}, \mathsf{USA}), \mathsf{Editor}(\mathsf{IEEE}, \mathsf{EU})\}.$$

∎

As a consequence, the issue of scalability of query answering with respect to large database instances turns out to be crucial [14, 41]. In this respect, an interesting approach is the one that aims at identifying *subclasses* of queries for which the problem is tractable [36, 35], or FOL-reducible [7, 47, 46].In particular, in [47] the authors study the problem for the class of conjunctive

queries, and define a subclass of CQs, called $\mathcal{C}_{tree}$, for which they provide an algorithm for FOL-rewriting under schemas which contain only KDs.

According to [47], $\mathcal{C}_{tree}$ is the class of conjunctive queries which respect the following three conditions:

(a) there not exist two or more atoms referring to the same relational symbol;

(b) every join appearing in the query body from non-key to key attributes involves the entire key of at least one relation;

(c) present an acyclic join graph.

The class $\mathcal{C}_{tree}$ is based on the notion of join graph.

Definition 6.3.4 (Join Graph) A *join graph* $G = \langle E, V \rangle$ of a conjunctive query q is the directed graph defined as follows:

(i) $N_i \in E$ for every atom a_i in the query body.

(ii) Given two nodes N_i and N_j, $(N_i, N_j) \in V$ if an existential shared variable x occurs in a non-key position in N_j and occurs also in N_i; the arc (N_i, N_j) will be labelled with the variable x.

(iii) Given a node N_i, $(N_i, N_i) \in E$ if an existential shared variable occurs at least twice in N_i, and at least one occurrence is in a non-key position.

We will indicate the join graph G of a query q with $JG(q)$. Given a join graph $JG(q)$ we indicate with $roots(JG(q))$ the set of roots of $JG(q)$, i.e., the set of nodes with no incoming arcs. Moreover, given a node N of a join graph $JG(q)$ of a query q we denote the set of successors of N with the symbol $jgsucc(N) = \{N' \mid (N, N') \in V\}$.

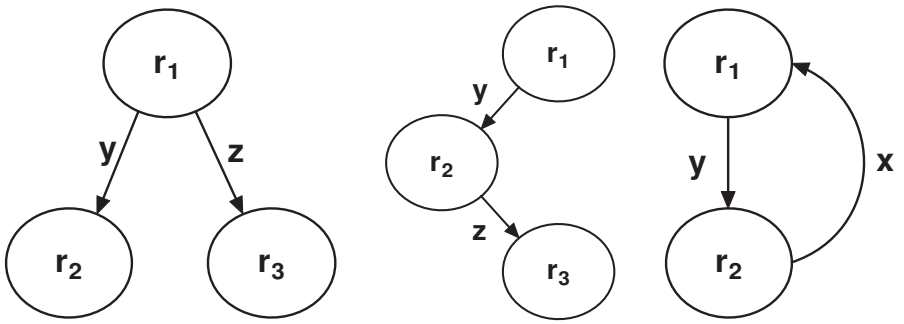
Example 6.3.5 Consider a database schema $\mathcal{S}$ containing a relation r_1 of arity 3, and two binary relations r_2, and r_3; the first argument of each such relation is a key argument. Let us consider now the following CQs:

$$\begin{array}{lcl} q_1 & :- & r_1(X, Y, Z) \wedge r_2(Y, W_1) \wedge r_3(Z, W_2). \\ q_2 & :- & r_1(X, Y, W_1) \wedge r_2(Y, Z) \wedge r_3(Z, W_2). \\ q_3 & :- & r_1(X, Y, Z) \wedge r_2(Y, X). \end{array}$$

The join graphs of q_1,q_2 and q_3 are depicted in Figure 6.1(a), Figure 6.1(b) and Figure 6.1(c) respectively. Notice that for readability of the figures, we labelled each arc with the corresponding join variable. Considering the query q_1 we have that $roots(JG(q_1)) = \{r_1\}$ and $jgsucc(r_1) = \{r_2, r_3\}$.

■

As pointed out in [47], this $\mathcal{C}_{tree}$ class of queries is very common, since cycles are rarely present in queries used in practice. However, no repeated symbols may occur in the queries, and



(a) Join graph of q_1 (b) Join graph of q_2 (c) Join graph of q_3

Figure 6.1: Join Graphs for queries of Example 6.3.5

query can have joins from non-key attributes only if such joins involve the entire key of at least one relation.

In this section we define a new class of CQs more general than $\mathcal{C}_{tree}$, called $\mathcal{C}^{+}_{tree}$, and propose an algorithm, called CQRewrite, for FOL-rewriting of such CQs. Conjunctive queries belonging to such a class respect condition (a) and (c) above, but admit joins from non-key attributes that not necessarily involve the entire key of a relation. In other words, condition (b) above has been removed. We now formally define the $\mathcal{C}^{+}_{tree}$ class of queries.

Definition 6.3.6 Let q be a conjunctive query whose join graph is the graph G. The query q belongs to the class $\mathcal{C}^{+}_{tree}$ if:

(i) there do not exist in the body of q two or more atoms referring to the same relational symbol;

(ii) the graph G is a forest.

Before defining an algorithm for first-order rewriting query in the $\mathcal{C}^{+}_{tree}$ class we need some preliminary definitions. First of all we propose a refined notion of join graph, in which we associate to each node an adornment which specifies the different nature of terms in the atoms. To do so, we will associate a *type* to each term of an atom according to the following definition.

Definition 6.3.7 (Type of a term) Let $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$ be a database schema, q be a CQ over $\mathcal{A}$, and $a = r(x_1, \ldots, x_n)$ be an atom (of arity n) occurring in the body of q. Then, let $key(r) = \{i_1, \ldots, i_k\}$ belong to $\mathcal{K}$, and let $1 \leq i \leq n$. The *type of the i-th argument of a in q*, denoted by $type(a, i, q)$ is defined as follows:

1. If $i_1 \leq i \leq i_k$, then:
   - if x_i is a head variable of q, a constant, or an existential shared variable, then $type(a, i, q) = KB$;
   - if x_i is an existential non-shared variable of q, then $type(a, i, q) = KU$.
2. Otherwise ($i \notin \{i_1, \ldots, i_k\}$):

- if x_i is a head variable of q or a constant, then $type(a, i, q) = B$;
- if x_i is an existential shared variable of q, then $type(a, i, q) = S$;
- if x_i is an existential non-shared variable of q, then $type(a, i, q) = U$.

Terms type by KB or B are called *bound terms*, otherwise they are called *unbound*.

We are now able to define the *typing* of an atom a of a CQ q, that will be used to adorn the corresponding node of the *typed join graph* of q. Informally, the typing of an atom a is an expression in which each attribute of a is associated with its type.

Definition 6.3.8 (Typing of an atom) Let $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$ be a database schema, q be a CQ over $\mathcal{A}$, and $a = r(x_1, \ldots, x_n)$ be an atom (of arity n) occurring in the body of q. The *typing* of an atom a of q is an expression of the form $r(x_1/t_1, \ldots, x_n/t_n)$ where each $t_i = type(a, i, q)$.

We now refine the notion of join graph, by labelling each node with the typing of the corresponding atom.

Definition 6.3.9 (Typed Join Graph) Given the join graph $G = \langle V, E \rangle$ of a conjunctive query q, the *typed join graph* of q is the graph obtained from G by labelling each node $N \in V$ with the typing of the corresponding atom with respect to the query q.

From now on we will refer always to typed join graph. Moreover, with a little abuse of notation, we will make use of new queries, called *typed queries*, in which each atom is substituted with its typing.

Definition 6.3.10 (Typed query) Given a database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$, and a conjunctive query q over $\mathcal{A}$, we define *typed query* the query q^t obtained by replacing each atom $a = r(x_1, \ldots, x_n)$ of q with its typing $r(x_1/t_1, \ldots, x_n/t_n)$.

Example 6.3.11 Let us consider again the database schema presented in Example 6.3.5 and the query

$$q(X, Z) \quad :- \quad r_1(X, Y, W_1) \wedge r_2(Y, Z).$$

The typed join graph of q is depicted in Figure 6.2. Moreover the typed query q^t obtained by typing the query q above is

$$q(X, Z) \quad :- \quad r_1(X/\mathbf{KB}, Y/\mathbf{S}, W_1/\mathbf{U}) \wedge r_2(Y/\mathbf{KB}, Z/\mathbf{B}).$$

■

The CQRewrite algorithm

We are now ready to define the algorithm CQRewrite, a FOL rewriting algorithm for CQs belonging to the $\mathcal{C}^+_{tree}$ class of queries tha. Notice that the new variables introduced by the algorithm NodeRewrite(see below) are always fresh symbols with respect to all the executions of the algorithm. We first define the algorithm then providing a detailed description of it.

The algorithm CQRewrite is the following:

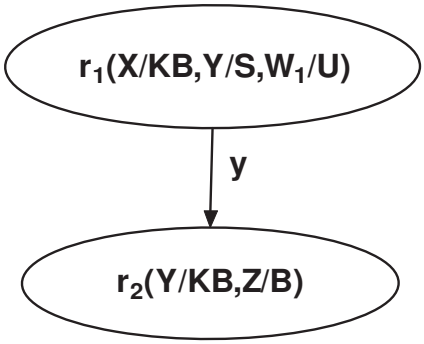


Figure 6.2: Typed join graph for query q of Example 6.3.11

Algorithm CQRewrite(q,$\mathcal{S}$)
Input: a first-order reducible CQ $q = h(x_1, \ldots, x_n) :\!- conj(x_1, \ldots, x_n, y_1, \ldots, y_m)$;
schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$
Output: FOL query (representing the rewriting of q)
begin
let $JG(q)$ be the labelled join graph of q;
return $\{ \bigwedge_{N \in roots(JG_{(q)})} \mathsf{NodeRewrite}(JG(q), N, q, \mathcal{S}) \}$
end

The above algorithm calls the algorithm NodeRewrite which computes the rewriting of the query q by exploiting the labelled join graph of q:
The algorithm recursively calls the subroutine NodeRewrite shown in Figure 6.3. Such subroutine is responsible for the generation of the first-order formula composing the rewriting of a CQ q. Informally, NodeRewrite recursively explores the join graph building up the rewriting of the query by producing a rewriting for each node N of the join graph. Such rewriting is constituted by the conjunction of two formulas f_1 and f_2: the former ensures that the query joins are satisfied, while the latter is responsible for checking the key dependencies.

We now provide an example of execution of the CQRewrite algorithm.

Example 6.3.12 Consider a database schema $\mathcal{S}$ containing two binary relation symbols $r/2$ and $s/2$. Let the first attribute of r and s be key attributes for those relations: we have that $\mathcal{K}$, i.e., the set of key dependencies of $\mathcal{S}$, contains the key dependencies $key(r) = \{1\}$ and $key(s) = \{1\}$.

Consider now the following conjunctive query q:

$$q :\!- r(X, Y), s(Y, Z).$$

whose typing is the expression

$$q :\!- r(X/\mathbf{KU}, Y/\mathbf{S}), s(Y/\mathbf{KB}, Z/\mathbf{U}).$$

The typed join graph of the query q (see Figure 6.4) contains two nodes N_1 and N_2 and an edge between them. The execution of the algorithm CQRewrite produces the following FOL formula:

Algorithm NodeRewrite$(JG(q), N, q, \mathcal{S})$
Input:
join graph $JG(q)$;
node N of $JG(q)$;
a typed conjunctive query $q = h(x_1, \ldots, x_n) \text{ :- } conj(x_1, \ldots, x_n, y_1, \ldots, y_m)$;
schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$;
Output: FOL formula
begin
let $a = r(x_1/t_1, \ldots, x_n/t_n)$ be the label of N;
for $i := 1$ **to** n **do**
if $t_i \in \{KB, B\}$ **then** $v_i := x_i$
else $v_i := y_i$, where y_i is a new variable;
if each argument of a is of type B or KB **then** $f_1 := r(x_1, \ldots, x_n)$
else begin
let $i_1, \ldots, i_m$ be the positions of the arguments of a of type S, U, KU;
$f_1 := \exists y_{i_1}, \ldots, y_{i_m}.\, r(v_1, \ldots, v_n)$
end;
if there exists no argument in a of type B or S
then return f_1
else begin
let $p_1, \ldots, p_c$ be the positions of the arguments of a of type U, S or B;
let $\ell_1, \ldots, \ell_h$ be the positions of the arguments of a of type B;
for $i := 1$ **to** c **do**
if $t_{p_i} = S$ **then** $z_{p_i} := x_{p_i}$ **else** $z_{p_i} := y_i''$, where y_i'' is a new variable
for $i := 1$ **to** n **do**
if $t_i \in \{KB, KU\}$ **then** $w_i := v_i$ **else** $w_i := z_i$;
$f_2 := \forall z_{p_1}, \ldots, z_{p_c}.\, r(w_1, \ldots, w_n) \rightarrow$

$$\left(\bigwedge_{N' \in jgsucc(N)} \mathsf{NodeRewrite}(JG(q), N', Q, \mathcal{S},) \wedge \bigwedge_{i \in \{\ell_1, \ldots, \ell_h\}} w_i = x_i \right) \vee$$

return $f_1 \wedge f_2$
end
end

Figure 6.3: The algorithm NodeRewrite

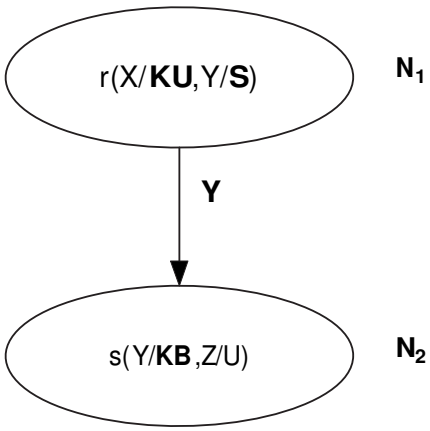


Figure 6.4: Typed join graph for query $q :\!- r(X,Y), s(Y,X)$.

$$\{ \ | \ \mathsf{NodeRewrite}(JG(q), N_1, q, \mathcal{S})\}$$

in which $\mathsf{NodeRewrite}(JG(q), N_1, q, \mathcal{S})$ is the following first-order formula:

$$\exists X, Y.r(X,Y) \wedge \forall Y'.r(X,Y') \rightarrow \mathsf{NodeRewrite}(JG(q), N_2, q, \mathcal{S})$$

In turn the recursive call $\mathsf{NodeRewrite}(JG(q), N_2, q, \mathcal{S})$ produces the following formula:

$$\exists Z.s(Y', Z)$$

Finally, the whole FOL rewriting of the query q is the following:

$$\{ \ | \ \exists X, Y.r(X,Y) \wedge \forall Y'.r(X,Y') \rightarrow \exists Z.s(Y', Z)\}$$

■

The answerCQ algorithm

Finally we define a query answering algorithm based on the above rewriting, which simply evaluates the query computed by $\mathsf{CQRewrite}(q,\mathcal{S})$ over the database instance $\mathcal{D}$. Formally:

Algorithm $\mathsf{answerCQ}(q,\mathcal{S},\mathcal{D})$
Input:
a first-order reducible CQ q;
a database schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$;
a database instance $\mathcal{D}$ for $\mathcal{S}$:
Output:
The evaluation of q over $\mathcal{D}$;
begin
$q_1 = \mathsf{CQRewrite}(q, S, D)$;
return ${q_1}^{\mathcal{D}}$;
end

Correctness of the answerCQ algorithm

We now prove soundness and completeness of the algorithm answerCQ. Without loss of generality, in such a proof (and also in the correctness proofs in Section 6.3.3 and Section 6.3.4), we restrict our attention to *boolean* queries, since the proof can be immediately generalized to the case of non-boolean queries.

In order to state the correctness of the answerCQ algorithm, we need some preliminary definitions. Notice that such definitions are provided for the general class of arbitrary unions of conjunctive queries.

Definition 6.3.13 (Image of a Query) Given a database $\mathcal{D}$ and a boolean UCQ q over the relational symbols of $\mathcal{D}$, we say that a set of facts $I \subseteq \mathcal{D}$ is an *image* of q in $\mathcal{D}$ if there exists a substitution σ of the variables occurring in q with the constants of $\mathcal{D}$ such that there exists a disjunct d_i of q for which $\sigma(d_i) = I$. Given a UCQ q and a database $\mathcal{D}$, $Images(q, \mathcal{D})$ denotes the set of all images I in $\mathcal{D}$ of the query q.

Definition 6.3.14 (Repairs of a database) Given a database $\mathcal{D}$ and a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, we define the set $Repairs(\mathcal{D}, \mathcal{S})$ as the set $\{R \mid R$ is a repair of $\mathcal{D}$ with respect to $\mathcal{K}\}$, i.e., $Repairs(\mathcal{D}, \mathcal{S})$ is the set of all the repairs of $\mathcal{D}$ with respect to the KDs in $\mathcal{K}$.

Definition 6.3.15 Given a set of KDs $\mathcal{K}$ and two facts f, f', we say that f' *is a $\mathcal{K}$-opponent to* f if f and f' are facts for the same relation r and their key arguments are equal. Furthermore, given a set of facts R, a set of KDs $\mathcal{K}$ and a fact f, we say that f *has a $\mathcal{K}$-opponent in R (or that f is a $\mathcal{K}$-opponent to R)* if there exists a fact $f' \in R$ such that f' is a $\mathcal{K}$-opponent to f.

Definition 6.3.16 Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a boolean CQ q, an image I of q, and a set of facts R, we denote by $First\text{-}Opponents(I, q, \mathcal{S}, R)$ the set of atoms of q where each atom a is such that the image of a in I has a $\mathcal{K}$-opponent in R and every predecessor b of a in the join graph of q is such that the image of b in I has no $\mathcal{K}$-opponent in R.

We now show a key property that will be used to prove soundness of the answerCQ algorithm.

Lemma 6.3.17 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database instance $\mathcal{D}$ for $\mathcal{S}$ and a boolean CQ $q \in \mathcal{C}^{+}_{tree}$, for each $R \in Repairs(\mathcal{D}, \mathcal{S})$, if the algorithm answerCQ$(q, \mathcal{S}, \mathcal{D})$ returns* `true` *and there exists $I \in Images(q, \mathcal{D})$ such that $a \in First\text{-}Opponents(I, q, \mathcal{S}, R)$, then there exists $I' \in Images(q, \mathcal{D})$ such that $First\text{-}Opponents(I', q, \mathcal{S}, R) = First\text{-}Opponents(I, q, \mathcal{S}, R) - \{a\} \cup M'$, where M' is a (non necessarily non-empty) set of atoms of q containing only successors of a in the join graph of q.*

Proof. Let a be an atom such that $a \in First\text{-}Opponents(I, q, \mathcal{S}, R)$ and let f' be a fact in R such that f' is a $\mathcal{K}$-opponent to the image of a in I. Since by hypothesis the algorithm returns `true`, there must exist an image I'' of q such that $f' \in I''$. Let I' be the set obtained from I'' by replacing the images of all the atoms that are not in the subtree of a (in the join graph of q) with the images in I of such atoms. It is easy to verify that the set I' above defined,

Algorithm computeConsAnswers$(Q, \mathcal{S}, \mathcal{D})$
Input:
a boolean UCQ Q,
a database schema $\mathcal{S}$ containing a set of key dependencies $\mathcal{K}$,
a database $\mathcal{D}$
Output:
`true/false`
begin
if there exists an image $I \in$ goodImages$(q, \mathcal{S}, \mathcal{D})$ of Q in $\mathcal{D}$ **then**
return `true`
else
return `false`
end

Figure 6.5: The computeConsAnswers algorithm

is an image of q; furthermore, by construction I' is such that *First-Opponents*$(I', q, \mathcal{S}, R) =$ *First-Opponents*$(I, q, \mathcal{S}, R) - \{a\} \cup M'$ where M' contains only successors of a (since for each atom a' of q, the images of a' in I and in I' are the same fact). □

The following lemma proves the soundness of the answerCQ algorithm.

Lemma 6.3.18 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean CQ $q \in \mathcal{C}^{+}_{tree}$, if the algorithm* answerCQ$(q, \mathcal{S}, \mathcal{D})$ *returns* `true`*, then, for every repair $R \in$ Repairs$(\mathcal{D}, \mathcal{S})$, q is true in R.*

Proof. Suppose the algorithm answerCQ$(q, \mathcal{S}, \mathcal{D})$ returns `true` and let $R \in$ *Repairs*$(\mathcal{D}, \mathcal{S})$. Then, there exists an image $I \in$ *Images*$(q, \mathcal{D})$, consequently, by Lemma 6.3.17 and by induction on the structure of the join graph of q, it follows that there exists an image $I' \in$ *Images*$(q, \mathcal{D})$ such that *First-Opponents*$(I', q, \mathcal{S}, R) = \emptyset$, which implies that $I' \subseteq R$, therefore q is true in R. □

We now turn our attention to the completeness of our algorithm. To this aim we define the computeConsAnswers algorithm (see Figure 6.5): given a UCQ Q over a database $\mathcal{D}$, whose schema $\mathcal{S}$ contains the set of key dependencies $\mathcal{K}$, the algorithm computeConsAnswers$(Q, \mathcal{S}, \mathcal{D})$ computes all the answers to Q over $\mathcal{D}$ that are consistent with respect to the set of key dependencies $\mathcal{K}$. The algorithm calls the algorithm goodImages (see Figure 6.6) that computes all the "good" images of a query Q over a database instance $\mathcal{D}$. Informally, the set of good images GI of q is a set of images of q in $\mathcal{D}$ such that, for each image $I \in GI$ and for every fact $f \in \mathcal{D}$ that is a $\mathcal{K}$-opponent to I, there exists an image $I' \in GI$ such that I' contains f.

We now state the correspondence between the algorithm computeConsAnswers and the algorithm answerCQ.

Algorithm goodImages$(Q, \mathcal{S}, \mathcal{D})$
Input:
a boolean UCQ Q,
a database schema $\mathcal{S}$ containing a set of key dependencies $\mathcal{K}$,
a database $\mathcal{D}$
Output:
the set of good images of the query Q in $\mathcal{D}$ under $\mathcal{K}$
begin
$F := \emptyset$;
$BI = \emptyset$;
$GI = Images(Q, \mathcal{D})$;
repeat
$GI' = GI$;
for each fact $f \in \mathcal{D}$ such that f does not occur in any image in GI **do**
$F := F \cup \{f\}$;
for each $I \in GI$ **do**
if there exists $f \in F$ such that f is a $\mathcal{K}$-opponents to I in $\mathcal{D}$
then $GI := GI - \{I\}$;
until $GI = GI'$;
return GI;
end

Figure 6.6: The goodImages algorithm

Lemma 6.3.19 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean CQ $q \in \mathcal{C}^{+}_{tree}$, the algorithm* answerCQ$(q, \mathcal{S}, \mathcal{D})$ *returns* `true` *iff the algorithm* computeConsAnswers$(q, \mathcal{S}, \mathcal{D})$ *returns* `true`*.*

The following lemma shows the completeness of the computeConsAnswers algorithm.

Lemma 6.3.20 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean CQ $q \in \mathcal{C}^{+}_{tree}$, if the algorithm* computeConsAnswers$(q, \mathcal{S}, \mathcal{D})$ *returns* `false`*, then there exists a repair $R \in Repairs(\mathcal{D}, \mathcal{S})$ such that q is false in R.*

Proof. Suppose computeConsAnswers(q,$\mathcal{S}$,$\mathcal{D}$) returns `false`. Then, the algorithm goodImages(q,$\mathcal{S}$,$\mathcal{D}$) returns an empty set. Now consider the set F computed by the algorithm goodImages(q,$\mathcal{S}$,$\mathcal{D}$) after the last execution of the repeat-until cycle. The set of facts F is such that the following property **P1** holds: for each $I \in Images(q, \mathcal{D})$ there exists $f \in F$ such that f is a $\mathcal{K}$-opponent to I. Now let f_1 and f_2 be any two facts belonging to Ff and such that f_1 and f_2 refer to the same relation symbol and f_1 and f_2 coincide on their key arguments. Then, it is easy to prove (by induction on the construction of F) the following property **P2**: for each image $I \in Images(q, \mathcal{D})$, if f_1 is a $\mathcal{K}$-opponent to I then also f_2 is a $\mathcal{K}$-opponent to I. Now let F_{clean} be any set of facts obtained from F by eliminating all key violations, i.e., F_{clean} is such that: (i) there is no pair of facts f_1 and f_2 in F_{clean} such that f_1 and f_2 refer to the same relation symbol and f_1 and f_2 coincide on their key arguments; (ii) for each fact $f \in F$ there exists a fact $f' \in F_{clean}$ such that $f' \in F$ and f' and f refer to the same relation symbol and coincide on their key arguments. Based on the above properties **P1** and **P2** it immediately follows that, for each $I \in Images(q, \mathcal{D})$, there exists $f \in F_{clean}$ such that f is a $\mathcal{K}$-opponent to I. Moreover, since F_{clean} satisfies the KDs in $\mathcal{K}$, it follows that there exists a repair $R \in Repairs(\mathcal{D}, \mathcal{S})$ such that $F_{clean} \subseteq R$. Consequently, since R contains a $\mathcal{K}$-opponent to each image of q in $\mathcal{D}$, it follows that q is false in R. □

Lemma 6.3.21 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean CQ $q \in \mathcal{C}^{+}_{tree}$, if the algorithm* answerCQ$(q, \mathcal{S}, \mathcal{D})$ *returns* `false`*, then there exists a repair $R \in Repairs(\mathcal{D}, \mathcal{S})$ such that q is false in R.*

Proof. The thesis follows immediately from Lemma 6.3.19 and Lemma 6.3.20. □

Theorem 6.3.22 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ and a boolean CQ $q \in \mathcal{C}^{+}_{tree}$, the algorithm* answerCQ$(q, \mathcal{S}, \mathcal{D})$ *returns* `true` *iff* `true` $\in$ $ConsAns(q, \mathcal{S}, \mathcal{D})$*.*

Proof. The thesis follows immediately from Lemma 6.3.18 and Lemma 6.3.21. □

6.3.3 Computing consistent answers of U-acyclic UCQs

In this section we start studying the first-order reducibility of union of conjunctive queries in the presences of key dependencies over the database schema. In detail, we propose a technique for

handling UCQs which is based on the treatment of the single queries that compose the union, according to the algorithm presented in the previous section. In the next section we will propose a further technique specifically tailored for unions, that is able to handle a broader class of UCQs. We believe that the study of this problem is important for two main reasons: (*i*) the capability to express unions is probably one of the main feature missed by single conjunctive queries and has practical relevance, and (*ii*) as we will explain better in Section 6.3.5, it is a necessary first step in order to extend the applicability of the first-order approach to *inclusion dependencies*, and hence, to *foreign keys*.

A possible approach to solve the problem of first-order rewriting UCQs in the presence of key dependencies, is to take advantage of the algorithm presented in Section 6.3.2 for single conjunctive queries. More precisely, given a UCQ Q of the form

$$h(x_1, \ldots, x_n) \coloneq d_1 \vee d_2 \vee \ldots \vee d_m$$

we simply apply the algorithm CQRewrite to every disjunct d_i of Q and take as result the query Q_f obtained by the union of the first-order queries produced by each single execution of the algorithm CQRewrite. In order to do that, we consider that every disjunct of Q is in the $\mathcal{C}^+_{tree}$ class. Formally, given a database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$ and a UCQ Q we provide the following algorithm UCQ-FolRewrite:

Algorithm UCQ-FolRewrite$(Q, \mathcal{S})$
Input: UCQ $Q = h(x_1, \ldots, x_n) \coloneq d_1 \vee \ldots \vee d_m$
such that $d_i \in \mathcal{C}^+_{tree}$ for $i \in \{1, \ldots, m\}$;
schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$
Output: FOL query
begin
for $i := 1$ **to** m **do**
begin
$q_i = h(x_1, \ldots, x_n) \coloneq d_i$;
compute $JG(q_i)$;
end
return $\{x_1, \ldots, x_n \mid \bigvee_{i=1}^{m} \bigwedge_{N \in roots(JG_{(q_i)})} \mathsf{NodeRewrite}(JG(q_i), N)\}$;
end

We now provide an example of execution of the algorithm.

Example 6.3.23 Consider a database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$, such that $\mathcal{A}$ contains the binary relation symbols r_1, r_2 and r_3, and $\mathcal{K}$ contains the dependencies $key(r_1) = \{1\}$, $key(r_2) = \{1\}$, $key(r_3) = \{1\}$. Consider the UCQ

$$Q \coloneq (r_1(x, y) \wedge r_2(y, z)) \vee (r_3(x, y) \wedge r_1(y, z))$$

over $\mathcal{A}$. The join graphs of each disjunct shown in Figure 6.7.

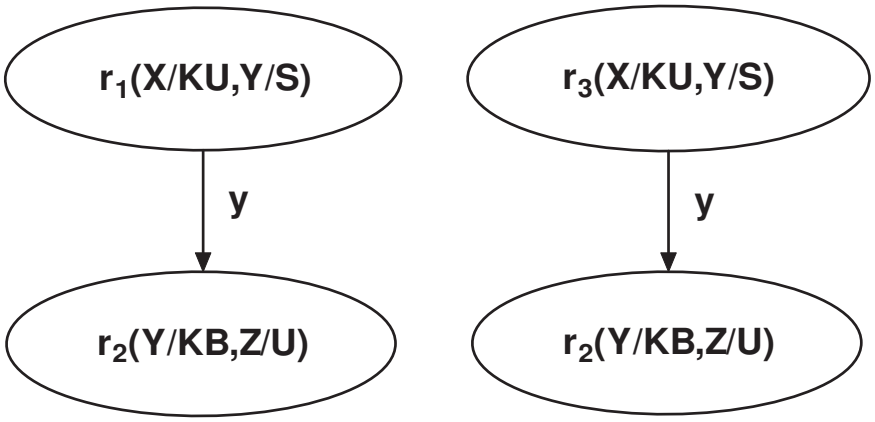


Figure 6.7: Typed join graph for the disjuncts of Q of Example 6.3.23

Now it is easy to see that any disjunct in the query is in class $\mathcal{C}^+_{tree}$. Then, the first-order query returned by the execution of UCQ-FolRewrite$(q, \mathcal{S})$ is

$$q_f = \{ \ | \ (\exists x, y.r_1(x, y) \wedge \forall y'.r_1(x, y') \rightarrow \exists z.r_2(y', z)) \vee \\ (\exists x, y.r_3(x, y) \wedge \forall z.r_3(x, z) \rightarrow \exists z.r_2(y', z))\}$$

For such an example it is possible to verify that the query above is actually the FOL-rewriting of the input query Q, i.e., for every database instance $\mathcal{D}$, $\vec{\mathbf{t}}$ is a consistent answer to q in $\mathcal{D}$ under $\mathcal{S}$ if and only if $\vec{\mathbf{t}} \in q_f^{\mathcal{D}}$. ∎

Now, the question arises whether the condition that any disjunct in the input query Q is in class $\mathcal{C}^+_{tree}$ is sufficient in order to guarantee soundness and, in particular, completeness of the algorithm. The following example shows that actually this is not the case.

Example 6.3.24 Assume to have a database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$, such that $\mathcal{A}$ contains the relation symbol r of arity 2, and $\mathcal{K}$ contains the dependency $key(r) = \{1\}$. Consider the UCQ $Q :- r(x, c_1) \vee r(x', c_2)$ over $\mathcal{A}$, in which c_1 and c_2 are different constant symbols. It is immediate to verify that any disjunct in the query is in class $\mathcal{C}^+_{tree}$. Then, the first-order query returned by the execution of UCQ-FolRewrite$(q, \mathcal{S})$ is

$$q_f = \{ \ | \ (\exists x, y.r(x, y) \wedge \forall y'.r(x, y') \rightarrow y' = c_1) \vee \\ (\exists x, y.r(x, y) \wedge (\forall y'.r(x, y') \rightarrow y' = c_2) \ \}$$

Now, assume to have the database instance $\mathcal{D} = \{r(a, c_1), r(a, c_2)\}$, which is not legal for $\mathcal{S}$. It is easy to see that $\mathcal{D} \not\models \Phi$, where Φ is the sentence corresponding to the body of q_f, i.e., according to a notation commonly adopted in the database theory for boolean queries, $\langle\rangle \notin \mathcal{D}$, where $\langle\rangle$ indicates the empty tuple. On the other hand, the repairs of $\mathcal{D}$ under $\mathcal{S}$ are $\mathcal{R}_1 = \{r(a, c_1)\}$ and $\mathcal{R}_2 = \{r(a, c_2)\}$, and the body of the query Q evaluates to true in both $\mathcal{R}_1$ and $\mathcal{R}_2$, i.e. $\langle\rangle$ is a consistent answer to Q in $\mathcal{D}$ under $\mathcal{S}$. ∎

The example above shows that the algorithm UCQ-FolRewrite is in general incomplete (even if it is easy to see that it is always sound). This is mainly due to the fact that separately rewriting single disjuncts does not take into account the interaction that may exist between them. Indeed, the body of the FOL-rewriting that the algorithm constructs for each single disjunct d_i (i.e., $\bigwedge_{N \in roots(JG(q_i))} \mathsf{NodeRewrite}(JG(q_i), N)$) is a FOL formula, which we denote with ϕ, such that, given an assignment of the free variables of ϕ (i.e., a tuple of constants $\vec{\mathbf{t}}$), the sentence $\phi(\vec{\mathbf{t}})$ is satisfied only by those database instances $\mathcal{D}$ such that in any repair of $\mathcal{D}$ there is an image of $\vec{\mathbf{t}}$ w.r.t the disjunct d_i. On the other hand, for a union of conjunctive queries q, for a tuple $\vec{\mathbf{t}}$ to be a consistent answer to q it is sufficient that in any repair of $\mathcal{D}$ there exists an image of the tuple w.r.t. q, i.e. with respect to *any* disjunct d_j of q (in other words, the disjunct which provides the image has not to be the same in any repair). This is actually the case we have in Example 6.3.24. Supporting the fact that uniting CQs that are FOL/rewritable may lead to non-FOL-rewritable UCQs, the following theorem formally shows that CQA for union of conjunctive queries is in general coNP-hard even if the queries that form the union are first-order rewritable.

Theorem 6.3.25 *Consistent answering of unions of $\mathcal{C}^+_{tree}$ conjunctive queries under key dependencies is coNP-hard.*

Proof. We reduce the problem of deciding whether a graph is not 3-colorable (which is a coNP-complete problem) to consistent answering of unions of $\mathcal{C}^+_{tree}$-CQs under key dependencies.

Given a graph $G = (V, E)$ (where V is the set of vertices and E is the set of edges, i.e., a binary relation over V), and a set $C = \{r, g, b\}$ (representing three colors), a *3-coloring for V* is a total function $\gamma : V \to C$, while a *3-coloring for G* is a 3-coloring for V such that, for each $(v_1, v_2) \in E$, $\gamma(v_1) \neq \gamma(v_2)$.

Let $\mathcal{I}$ be the database schema over the relational signature $edge/2$, $color_1/2$, $color_2/2$ containing the following key dependencies:

$$key(color_1) = \{1\}$$
$$key(color_2) = \{1\}$$

Now let q be the following boolean UCQ:

$$\begin{aligned} q \quad :- \quad & edge(X,Y), color_1(X,g), color_2(Y,g) \vee \\ & edge(X,Y), color_1(X,b), color_2(Y,b) \vee \\ & edge(X,Y), color_1(X,r), color_2(Y,r) \vee \\ & color_1(X,g), color_2(X,b) \vee \\ & color_1(X,g), color_2(X,r) \vee \\ & color_1(X,b), color_2(X,r) \end{aligned}$$

Observe that each conjunction in q is a query in $\mathcal{C}^+_{tree}$, since there are only key-to-key joins, and there are no repetitions of relations, hence q is a union of $\mathcal{C}^+_{tree}$-CQs.

Finally, given a graph $G = (V, E)$, we define the following database instance:

$$\begin{aligned} \mathcal{D} \quad = \quad & \{edge(v_1, v_2) \mid (v_1, v_2) \in E\} \cup \\ & \{color_1(v, c) \mid v \in V \text{ and } c \in C\} \cup \\ & \{color_2(v, c) \mid v \in V \text{ and } c \in C\} \end{aligned}$$

We now prove that q is true in all repairs of $(\mathcal{I}, \mathcal{D})$ if and only if there exists no 3-coloring for G.

($\Rightarrow$): Suppose there exists a 3-coloring γ for G. Then, γ is such that, for each $(v_1, v_2) \in E$, $\gamma(v_1) \neq \gamma(v_2)$. Now let

$$\begin{aligned} \mathcal{R} \quad = \quad & \{edge(v_1, v_2) \mid (v_1, v_2) \in E\} \cup \\ & \{color_1(v, c) \mid c = \gamma(v)\} \cup \\ & \{color_2(v, c) \mid c = \gamma(v)\} \end{aligned}$$

It is immediate to verify that $\mathcal{R}$ is a repair of $(\mathcal{I}, \mathcal{D})$, since $\mathcal{R}$ satisfies the key dependencies in $\mathcal{I}$ and, for every fact f such that $f \in \mathcal{D} - \mathcal{R}$, $\mathcal{R} \cup \{f\}$ violates a key dependency (because there is a fact f' in $\mathcal{R}$ with the same key as f). Moreover, the first three disjuncts of q are false in $\mathcal{R}$, due to the hypothesis that C is a 3-coloring for G, and the last three disjuncts of q are false in $\mathcal{R}$, because the extensions of $color_1$ and $color_2$ are the same in $\mathcal{R}$ and correspond to the function C. Consequently, q is false in $\mathcal{R}$.

($\Leftarrow$): Suppose $\mathcal{R}'$ is a repair of $(\mathcal{I}, \mathcal{D})$ such that q is false in $\mathcal{R}'$. First, due to the presence of the key dependencies in $\mathcal{I}$, and to the definition of repair, in $\mathcal{R}'$ both $color_1$ and $color_2$ are total functions from V to C. Then, since each of the three last disjuncts in q is false in $\mathcal{R}'$, it follows that in $\mathcal{R}'$ there is no vertex v such that the values of $color_1$ and $color_2$ (interpreted as functions) on v disagree, which implies that $color_1$ and $color_2$ are the same function, and in particular they correspond to a 3-coloring γ' for V. Finally, since each of the first three disjuncts of q are false in $\mathcal{R}'$, it follows that, for each $(v_1, v_2) \in E$, $\gamma'(v_1) \neq \gamma'(v_2)$, therefore γ' is a 3-coloring for G. □

Despite the above limitations of the algorithm, we are able to identify a subclass of union of conjunctive queries for which the algorithm UCQ-FolRewrite is sound and complete. To this aim, we define the class of U-acyclic UCQs. This class of queries is based on the notion of UCQ-graph defined below.

Definition 6.3.26 The UCQ-graph of a boolean UCQ Q is a graph such that

- There is a node N for every CQ $q \in Q$;
- there is an edge from a node N_i (associated to a CQ $q_i \in Q$) to a node N_j (associated to a CQ $q_j \in Q$), if q_i contains an atom a with at least a non-key bound attribute and q_j contains an atom b, with the same relational symbol of a, such that
  - key arguments of a and b are unifiable;
  - there is at least a non-key attribute in which a e b do not contain the same constant symbol.
  - if the nodes a and b are both roots in the join graphs of q_i and q_j, then the edge is labelled by r, where r is the relation symbol occurring in a and b.

We are now able to formally characterize the class of U-acyclic UCQs.

Definition 6.3.27 A UCQ Q is U-acyclic if the UCQ-graph of Q is acyclic.

We point out that the class of U-acyclic UCQs is first-order reducible, and for queries in this class the algorithm UCQ-FolRewrite is sound and complete.

The answerUCQA algorithm

We now define a query answering algorithm for UCQs belonging to the class of U-acyclic unions of conjunctive queries, based on the above rewriting, which simply evaluates the query computed by UCQ-FolRewrite(q,$\mathcal{S}$) over the database instance $\mathcal{D}$. Formally:

Algorithm answerUCQA(Q,$\mathcal{S}$,$\mathcal{D}$)
Input:
a U-acyclic UCQ Q;
a database schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$;
a database instance $\mathcal{D}$ for $\mathcal{S}$:
Output:
The evaluation of Q over $\mathcal{D}$;
begin
$Q_1 = \mathsf{UCQ\text{-}FolRewrite}(Q, S)$;
return ${Q_1}^{\mathcal{D}}$;
end

Correctness of the answerUCQA algorithm

We now prove soundness and completeness of the answerUCQA.

Lemma 6.3.28 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean UCQ Q, if the algorithm answerUCQA$(Q, \mathcal{S}, \mathcal{D})$ returns* `true`*, then, for every repair $R \in$ Repairs$(\mathcal{D}, \mathcal{S})$, Q is true in R.*

Proof. Let Q be the union of the boolean CQs $q_1, \ldots, q_n$. The thesis immediately follows from Lemma 6.3.18 and from the following facts: (i) by definition of answerUCQA, answerUCQA$(Q, \mathcal{S}, \mathcal{D})$ returns `true` iff there exists i such that $1 \leq i \leq n$ and answerCQ$(q_i, \mathcal{S}, \mathcal{D})$ returns `true`; (ii) $ConsAns(Q) \supseteq \bigcup_{i \in \{1,\ldots,n\}} ConsAns(q_i)$. □

Lemma 6.3.29 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean UCQ Q such that Q is UCQ-acyclic, if the algorithm answerUCQA$(q, \mathcal{S}, \mathcal{D})$ returns* `false`*, then there exists a repair $R \in$ Repairs$(\mathcal{D}, \mathcal{S})$ such that q is false in R.*

Proof. The thesis is an immediate consequence of Lemma 6.3.34 which will be presented in the next section. □

We are now able to state the following result:

Theorem 6.3.30 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{D}$ and a boolean UCQ Q such that Q is UCQ-acyclic, the algorithm answerUCQA$(q, \mathcal{S}, \mathcal{D})$ returns* `true` *iff* `true` $\in ConsAns(q, \mathcal{S}, \mathcal{D})$.

Proof. The thesis follows immediately from Lemma 6.3.28 and Lemma 6.3.29. □

6.3.4 Computing consistent answers for weakly-U-cyclic UCQs

We now try to extend the applicability of the rewriting technique presented in the previous section, by defining a new FOL-rewriting algorithm specifically tailored for such *weakly-U-cyclic* unions of conjunctive queries, a class of UCQs which extends the class of U-acyclic UCQs previously defined. We first define such queries and then provide the algorithm for FOL-rewriting of queries in that class.

Definition 6.3.31 A UCQ Q is weakly-U-cyclic if the UCQ-graph of Q is such that every cycle in the graph is such that all edges in the cycle are labelled and have the same label.

The idea is to modify the algorithm UCQ-FolRewrite presented above in order to deal with a special form of cycles in the UCQ-graph of the query. The resulting algorithm is called UCQ-FolRewriteNew and is presented below.

Algorithm UCQ-FolRewriteNew(Q,$\mathcal{S}$)
Input: a weakly-U-cyclic UCQ $Q = h(x_1, \ldots, x_n) \text{ :- } d_1 \vee \ldots \vee d_m$;
schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$
Output: FOL query (representing the rewriting of Q)
begin
let Q^t be the typed query associated to Q;
let UG be the UCQ-graph of Q^t;
for $i := 1$ **to** m **do**
let d_i^t be the typed disjunct associated to d_i;
return $\{x_1, \ldots, x_n \mid \bigvee_{i=1,\ldots,m} \mathsf{DisjunctRewrite}(d_i^t, Q^t, UG, \mathcal{S}, \emptyset)\ \}$
end

The above algorithm calls the following algorithm DisjunctRewrite, which computes the rewriting of a single disjunct d_i of the UCQ:

Algorithm DisjunctRewrite($d, Q, UG, \mathcal{S}, \mathcal{P}$)
Input:
a typed union of conjunctive queries $Q = h(x_1, \ldots, x_n) \text{ :- } d_1 \vee \ldots \vee d_m$;
a typed disjunct d that appears in Q;
the UCQ-graph UG of Q;
a schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$;
a set $\mathcal{P}$ of disjuncts of Q;
Output: FOL query (representing the rewriting of the disjunct d)
begin
$q = h(x_1, \ldots, x_n) \text{ :- } d$;
compute $JG(q)$;
return $\{ \bigwedge_{N \in roots(JG_{(q)})} \mathsf{NodeRewriteNew}(JG(q), N, q, UG, \mathcal{S}, \mathcal{P}) \}$
end

Algorithm NodeRewriteNew$(JG(q), N, q, UG, \mathcal{S}, \mathcal{P})$
Input:
join graph $JG(q)$;
node N of $JG(q)$;
a typed conjunctive query $q = h(x_1, \ldots, x_n) :\!- conj(x_1, \ldots, x_n, y_1, \ldots, y_m)$;
a UCQ-graph UG;
schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K} \rangle$;
a set $\mathcal{P}$ of disjuncts of Q;
Output: FOL formula
begin
if $q \notin \mathcal{P}$ **or** N is not root in JG(q) **then**
begin
let $a = r(x_1/t_1, \ldots, x_n/t_n)$ be the label of N;
for $i := 1$ **to** n **do**
if $t_i \in \{KB, B\}$ **then** $v_i := x_i$
else $v_i := y_i$, where y_i is a new variable;
if each argument of a is of type B or KB **then** $f_1 := r(x_1, \ldots, x_n)$
else begin
let $i_1, \ldots, i_m$ be the positions of the arguments of a of type S, U, KU;
$f_1 := \exists y_{i_1}, \ldots, y_{i_m}.\, r(v_1, \ldots, v_n)$
end;
if there exists no argument in a of type B or S
then return f_1
else begin
let $p_1, \ldots, p_c$ be the positions of the arguments of a of type U, S or B;
let $\ell_1, \ldots, \ell_h$ be the positions of the arguments of a of type B;
for $i := 1$ **to** c **do**
if $t_{p_i} = S$ **then** $z_{p_i} := x_{p_i}$ **else** $z_{p_i} := y_i''$, where y_i'' is a new variable
for $i := 1$ **to** n **do**
if $t_i \in \{KB, KU\}$ **then** $w_i := v_i$ **else** $w_i := z_i$;
$\mathcal{P} = \mathcal{P} \cup \{q\}$;
$f_2 := \forall z_{p_1}, \ldots, z_{p_c}.\, r(w_1, \ldots, w_n) \rightarrow$

$$\left(\bigwedge_{N' \in jgsucc(N)} \mathsf{NodeRewriteNew}(JG(q), N', q, UG, \mathcal{S}, \mathcal{P}) \wedge \bigwedge_{i \in \{\ell_1, \ldots, \ell_h\}} w_i = x_i \right) \vee$$

$$\bigvee_{d_j \in labelledSucc(q, UG)} \exists u_{j_1}, \ldots, u_{j_s}. w_1 = u_1 \wedge \ldots \wedge w_n = u_n \wedge \mathsf{DisjunctRewrite}(\tau(d_j), \tau(Q), UG, \mathcal{S}, \mathcal{P});$$

return $f_1 \wedge f_2$
end
end
end
else
return $\top$
end

Figure 6.8: The algorithm NodeRewriteNew

The subroutine NodeRewriteNew is presented in Figure 6.8.

In the algorithm DisjunctRewrite and NodeRewriteNew we make use of the data structure $\mathcal{P}$, corresponding to a set of disjuncts which represents the nodes of the UCQ-graph of Q that have been already processed (and should not be re-processed again - see proof of Lemma 6.3.32): indeed, the algorithm NodeRewriteNew is executed over a disjunct q only if q has not been already added to the set $\mathcal{P}$ (see the first if-then-else statement of the algorithm).

Furthermore, $labelledSucc(q, UG)$ denotes the set of disjuncts of Q that are successors of the disjunct q in the UCQ-graph of Q (which in both algorithm is represented by UG) through a labelled edge; $u_1, \ldots, u_n$ denote the terms of the atom a of the node N and $u_{i_1}, \ldots, u_{i_s}$ are the variable terms occurring in such atom. The operator τ modifies the typing of each atom by assigning the type KB to the key argument of the interacting atom, and the B type to the other, non-key arguments.

The answerUCQW algorithm

Finally, we define a query answering algorithm for UCQs belonging to the class of weakly-U-cyclic unions of conjunctive queries, based on the above rewriting, which simply evaluates the query computed by UCQ-FolRewriteNew(q,$\mathcal{S}$) over the database instance $\mathcal{D}$. Formally:

Algorithm answerUCQW(Q,$\mathcal{S}$,$\mathcal{D}$)
Input:
a weakly-U-cyclic UCQ Q;
a database schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$;
a database instance $\mathcal{D}$ for $\mathcal{S}$:
Output:
the evaluation of Q over $\mathcal{D}$;
begin
Q_1 = UCQ-FolRewriteNew(Q, S);
return ${Q_1}^{\mathcal{D}}$;
end

Correctness of FOL rewriting for weakly-U-cyclic UCQs

We start by proving soundness of the algorithm answerUCQW.

Lemma 6.3.32 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean UCQ Q, if the algorithm answerUCQW($Q, \mathcal{S}, \mathcal{D}$) returns* `true`*, then, for every repair $R \in Repairs(\mathcal{D}, \mathcal{S})$, Q is true in R.*

Proof. Suppose answerUCQW($Q, \mathcal{S}, \mathcal{D}$) returns `true`. Then, there exists an image $I \in Images(Q, \mathcal{D})$. Let $R \in Repairs(\mathcal{D}, \mathcal{S})$ and let q be the conjunction of Q of which I is an image. Now suppose there exists $f \in \mathcal{D}$ such that f is a $\mathcal{K}$-opponent to I: without loss of generality we can assume that f is the image of an atom a in q such that $a \in First\text{-}Opponents(I, q, \mathcal{S}, R)$. Since the algorithm returns `true`, there exists $I' \in Images(q, \mathcal{D})$ such that $f \in I'$. There are two possible cases:

1) I' is an image of the same conjunction q in Q of which I is an image. Then, by a proof analogous to the one of Lemma 6.3.17 we can conclude that, for each $R \in \mathit{Repairs}(\mathcal{D}, \mathcal{S})$, $\mathit{First\text{-}Opponents}(I, q, \mathcal{S}, R) = \mathit{First\text{-}Opponents}(I', q, \mathcal{S}, R) - \{a\} \cup M'$ where M' is a set of atoms which are successors of a in the join graph of q;

2) I' is an image of a conjunction q' in Q different from q. This implies that there exists an edge from q to q' in the UCQ-graph of Q (in fact, since $a \in \mathit{First\text{-}Opponents}(I, q, \mathcal{S}, R)$, q and q' must contain atoms with the same relation symbol and whose arguments unify according to the construction of edges in the UCQ-graph).

Now, if there are $\mathcal{K}$-opponents to I' in $\mathcal{D}$ we have to repeat the above step, i.e., if there exists $f' \in \mathcal{D}$ such that f' is a $\mathcal{K}$-opponent to I', then there exists $I'' \in \mathit{Images}(q, \mathcal{D})$ such that $f' \in I''$ and the two cases 1) and 2) are now possible for I''. However, notice that in case 2), we can assume that *I' is not an image of the query q of which I was an image.* In fact, due to the form of the cycles in the UCQ-graph of Q (which only involve root atoms in the join graphs of the queries), it follows that if I'' is an image of q, then the fact f is also an opponent to I'', so this image has already been ruled out by the fact f and should not be taken into consideration anymore. Based on the above property, we can conclude that, in the iteration of the above step, the case 2) can be considered at most n times for n images (where n is the number of conjunctions in Q). In other words, after the n-th "jump" to a different conjunction of Q, we can apply a proof analogous to the one of Lemma 6.3.18 and we can conclude that there exists an image I of a conjunction q in Q such that Q is true in R. □

We then state a property analogous to Lemma 6.3.19 for the algorithm answerUCQW.

Lemma 6.3.33 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean UCQ Q such that Q is weakly-U-cyclic, the algorithm answerUCQW$(q, \mathcal{S}, \mathcal{D})$ returns `true` iff the algorithm computeConsAnswers$(q, \mathcal{S}, \mathcal{D})$ returns `true`.*

Based on the above lemma, we can prove completeness of the algorithm answerUCQW.

Lemma 6.3.34 *Given a relational schema $\mathcal{S}$ containing the set of key dependencies $\mathcal{K}$, a database $\mathcal{D}$ for $\mathcal{S}$ and a boolean UCQ Q such that Q is weakly-U-cyclic, if the algorithm answerUCQW$(q, \mathcal{K}, \mathcal{D})$ returns `false`, then there exists a repair $R \in \mathit{Repairs}(\mathcal{D}, \mathcal{K})$ such that q is false in R.*

Proof. The thesis is an immediate consequence of Lemma 6.3.20 and Lemma 6.3.33. □

Theorem 6.3.35 *Given a set of KDs $\mathcal{K}$, a database $\mathcal{D}$ and a boolean UCQ Q such that Q is weakly UCQ-acyclic, the algorithm answerUCQW$(q, \mathcal{K}, \mathcal{D})$ returns `true` iff `true` $\in \mathit{ans}(q, \mathcal{K}, \mathcal{D})$.*

Proof. The thesis follows immediately from Lemma 6.3.32 and Lemma 6.3.34. □

6.3.5 First-order reducibility of CQs under KDs and IDs

In this section we study FOL-reducibility of CQA for unions of conjunctive queries under key and inclusion dependencies. Notice that, due to the wide applicability range of this kind of constraints, the possibility to efficiently handle KDs and IDs (and hence *foreign key*) is an important outcome of this work. We adopt the loosely-sound semantics (see Section 6.3.1) in order to deal with both KDs and IDs: this allows for the repairs to be obtained by adding as well as removing tuples in order to repair inconsistent instances.

In the following we first recall some basic results on query answering under KDs and IDs, then providing a sound and complete algorithm for first-order rewriting of UCQs in the presence of the aforementioned constraints.

Decidability of CQA under KDs and IDs

As shown in [19], the problem of computing the consistent answers of a conjunctive query issued over a database schema enriched with key and inclusion dependencies is in general undecidable. Anyway, the authors of that work have shown that for *non-key-conflicting* (NKC) database schemas, i.e., database schemas with particular combination of KDs and IDs, the problem is decidable. For the sake of completeness, we report below the formal definition of NKC schema.

Definition 6.3.36 Given a database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{F} \rangle$, an inclusion dependency in $\mathcal{F}$ of the form $r_1[\mathbf{A}_1] \subseteq r_2[\mathbf{A}_2]$ is a *non-key-conflicting inclusion dependency* (NKCID) with respect to $\mathcal{K}$ if either: (i) no KD is defined on r_2, or (ii) the KD $key(r_2) = \mathbf{K}$ is in $\mathcal{K}$, and $\mathbf{A}_2$ is not a strict superset of $\mathbf{K}$, i.e., $\mathbf{A}_2 \not\supset \mathbf{K}$. Moreover, the schema $\mathcal{S}$ is *non-key-conflicting* (NKC) if all the IDs in $\mathcal{F}$ are NKCIDs with respect to $\mathcal{K}$.

In other words, a set of dependencies is NKC if no ID in $\mathcal{F}$ propagates a proper subset of the key of the relation in its right-hand side. Notice that the class of NKCIDs comprises the well-known class of foreign key dependencies.

As a further result, the authors showed that for NKC database schemas a separation theorem holds that allows for computing the consistent answers to be computed in a modular way, by repairing first IDs and then KDs.

The ID-rewrite algorithm

According to the loosely-sound semantics, when a database instance does not satisfy a set of inclusion dependencies, we say that such instance is *incomplete*. In [20] the authors presented ID-rewrite, an algorithm to deal with IDs in the presence of incomplete information. Informally, ID-rewrite exploits IDs in order to produce the *perfect rewriting* of a UCQ q. The output of the algorithm is a UCQ whose evaluation over the inconsistent and incomplete database instance provides the certain answers.

In order to present ID-rewrite we need some preliminary definitions.

Definition 6.3.37 Given an atom $g = s(X_1, \ldots, X_n)$ and an inclusion $I = r[i_1, \ldots, i_k] \subseteq s[j_1, \ldots, j_k]$, we say that I *is applicable to* g if, for each ℓ such that $1 \leq \ell \leq n$, if $X_\ell \neq \_$

then there exists h such that $j_h = \ell$. Moreover, we denote with $gr(g, I)$ the atom $s(Y_1, \ldots, Y_m)$ (m is the arity of s in $\mathcal{S}$) where for each ℓ such that $1 \leq \ell \leq m$, $Y_\ell = X_{j_h}$ if there exists h such that $i_h = \ell$, otherwise $Y_\ell =\_$.

Roughly speaking, an inclusion I is applicable to an atom g if the relation symbol of g corresponds to the symbol in the right-hand side of I and if all the attributes for which *bounded* terms appear in g are propagated by the inclusion I. We recall that, given a CQ q, a bound term is a head-variable, a join variable or a constant. When I is applicable to g, $gr(g, I)$ denotes the atom obtained from g by using I as a rewriting rule whose direction is right-to-left.

Definition 6.3.38 Given an atom $g_1 = r(X_1, \ldots, X_n)$ and an atom $g_2 = r(Y_1, \ldots, Y_n)$, we say that g_1 *and* g_2 *unify* if for each i such that $1 \leq i \leq n$, either $X_i = Y_i$ or $X_i =\_$ or $Y_i =\_$. Moreover, if g_1 and g_2 unify, we denote as $U(g_1, g_2)$ the atom $r(Z_1, \ldots, Z_n)$ where, for each i, if $X_i = Y_i$ or $Y_i =\_$ then $Z_i = X_i$, otherwise $Z_i = Y_i$.

Informally, two atoms unify if they can be made equal through a substitution of each instance of the special symbol ξ with other terms.

Algorithm ID-rewrite($\mathcal{S}, q$)
Input: relational schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{F} \rangle$,
union of conjunctive queries q
Output: perfect rewriting of q
$Q' := Q$;
repeat
  $Q_{aux} := Q'$;
  for each $q \in Q_{aux}$ **do**
  (a) **for each** $g_1, g_2 \in body(q)$ **do**
    if g_1 and g_2 unify
    then $Q' := Q' \cup \{\tau(reduce(q, g_1, g_2))\}$;
  (b) **for each** $g \in body(q)$ **do**
    for each $I \in \mathcal{F}$ **do**
      if I is applicable to g
      then $Q' := Q' \cup \{\, q[g/gr(g, I)] \,\}$
until $Q_{aux} = Q'$;
return Q'

Figure 6.9: The ID-rewrite algorithm

In Figure 6.9 we present the algorithm ID-rewrite. Informally, the algorithm computes the closure of the set of conjunctive queries Q with respect to the following two rules:

(i) if there exists a query $q \in Q$ such that $body(q)$ contains two atoms g_1 and g_2 that unify, then the algorithm computes the query $reduce(q, g_1, g_2)$, which is obtained from q by replacing

g_1 and g_2 with $U(g_1, g_2)$ in the query body, and then by applying the substitution obtained in the computation of $U(g_1, g_2)$ to the whole query. Such a new query is then transformed by the function τ, which replaces with _ each variable symbol X such that there is a single occurrence of X in q. The use of τ is necessary in order to guarantee that each *unbounded* variable is represented by the symbol _. Such a query is then added to Q.

(*ii*) if there exists an inclusion I and a query $q \in Q$ containing an atom g such that I is applicable to g, then the algorithm adds to Q the query obtained from q by replacing g with $gr(g, I)$ in its body (denoted in the algorithm as $q[g/gr(g, I)]$). Namely, this step adds new conjunctions obtained by applying inclusion dependencies as rewriting rules (applied from right to left).

The above rules correspond respectively to steps (a) and (b) of the algorithm.

In [20] termination and correctness of ID-rewrite are shown.

The FolKIDRewrite algorithm

In this section we present the FolKIDRewrite algorithm, that is, an algorithm for computing the first-order rewriting of a (union of) conjunctive queries in the presence of KDs and IDs. When evaluating the first-order query obtained by the execution of FolKIDRewrite over an inconsistent and incomplete database instance, we obtain only the consistent answers to the original query.

Algorithm FolKIDRewrite($\mathcal{S}, Q$)
Input: a NKC relational schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{F} \rangle$,
a (union of) conjunctive queries Q
Output: `fail` or first-order rewriting of Q
$\mathcal{S}' = \langle \mathcal{A}, \mathcal{F} \rangle$;
$Q' :=$ ID-rewrite($\mathcal{S}', Q$);
if Q' **is** weakly-U-cyclic **then**
  return UCQ-FolRewriteNew(Q',$\mathcal{S}$);
else
  `fail`

Figure 6.10: The FolKIDRewrite algorithm

The algorithm exploits the separation property proved in [19] and first consider IDs and then KDs. In particular it first rewrite the input query Q by means of the ID-rewrite algorithm, and then apply the algorithm UCQ-FolRewriteNew presented in Section 6.3.3. Notice that a necessary condition for this procedure to work correctly, is that the rewriting produced by ID-rewrite is in the class of weakly-U-cyclic queries. The pseudo-code of the algorithm is shown in Figure 6.10. The following theorem states the correctness of the algorithm and can be easily proven exploiting the results of [19] and those presented in the previous section.

Theorem 6.3.39 *Let $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{F} \rangle$ be a database schema, Q be a (union of) conjunctive query over $\mathcal{S}$. Let $Q_r =$ FolKIDRewrite($\mathcal{S}, Q$) be the first-order rewriting of Q. Then, for every database*

instance $\mathcal{D}$ for $\mathcal{S}$, a tuple t is a consistent answer to Q in $\mathcal{D}$ under $\mathcal{S}$ iff $t \in Q_r^{\mathcal{D}}$.

6.3.6 First-order reducibility of CQs under KDs and EDs

In this section we study CQA for conjunctive queries under key dependencies and exclusion dependencies [69]. As already mentioned earlier in this chapter, the problem of answering conjunctive queries under key dependencies is in general intractable.

The following theorem show coNP-hardness also for the problem of computing consistent answers to conjunctive queries over inconsistent databases in the presence of only exclusion dependencies (EDs) (under the repair semantics introduced in Section 6.3.1)[1].

Theorem 6.3.40 *Let $\mathcal{S} = \langle \mathcal{A}, \emptyset, \mathcal{E} \rangle$ be a database schema containing only EDs, $\mathcal{D}$ a database instance for $\mathcal{S}$, q a CQ of arity n over $\mathcal{S}$, and t an n-tuple of constants. The problem of establishing whether t is a consistent answer to q in $\mathcal{D}$ under $\mathcal{S}$ is coNP-hard with respect to data complexity.*

Proof (sketch). We prove coNP-hardness by reducing the 3-colorability problem to the complement of our problem. Consider a graph $G = \langle V, E \rangle$ with a set of vertices V and edges E. We define a relational schema $\mathcal{S} = \langle \mathcal{A}, \emptyset, \mathcal{E} \rangle$ where $\mathcal{A}$ consists of the relation *edge* of arity 2, and the relation *col* of arity 5, and $\mathcal{E}$ contains the dependencies $col[3] \cap col[4] = \emptyset$, $col[3] \cap col[5] = \emptyset$, $col[4] \cap col[5] = \emptyset$. The instance $\mathcal{D}$ is defined as follows:

$$\begin{aligned} \mathcal{D} \quad = \quad & \{col(n, 1, n, \_, \_), col(n, 2, \_, n, \_), col(n, 3, \_, \_, n) | \\ & n \in V\} \cup \{edge(x, y) | \langle x, y \rangle \in E\}. \end{aligned}$$

Where each occurrence of the meta-symbol $\_$ denotes a different constant not occurring elsewhere in the database. Intuitively, to represent the fact that vertex $n \in V$ is assigned with color $i \in \{1, 2, 3\}$, $\mathcal{D}$ assigns to *col* a tuple in which i occurs as second component and n occurs as first and also as $2 + i$-th component. The EDs of $\mathcal{S}$ impose that consistent instances assign no more than one color to each node. Finally, we define the query

$$q \quad \leftarrow \quad edge(x, y), col(x, z, w_1, w_2, w_3), col(y, z, w_4, w_5, w_6).$$

On the basis of the above construction it is possible to show that G is 3-colorable (i.e., for each pair of adjacent vertices, the vertices are associated with different colors) if and only if the empty tuple $\langle\rangle$ is not a consistent answer to q in $\mathcal{D}$ under $\mathcal{S}$ (i.e., the boolean query q has a negative answer). □

Datalog$^\neg$ Rewriting

Before studying FOL-reducibility of conjunctive queries under KDs and EDs we provide a sound and complete rewriting technique for consistent query answering in the presence of such constraints. To this aim, we make use of Datalog$^\neg$, i.e., Datalog enriched with (unstratified) negation, under stable model semantics [42]. Although it could seem out of the scope of this

[1] We consider the decision problem associated to query answering (see e.g., [35])

section, it is important to show here such a rewriting technique: indeed, a similar result has never been presented in the literature. On the other hand, we will use such general procedure in order to provide an experimental validation of the optimization achieved by means of the first-order rewriting.

From a computational point of view, Datalog$^\neg$ is coNP-complete with respect to data complexity, and therefore is well suited for dealing with the high computational complexity of our problem. The rewriting that we present in the following extends the one proposed in [19] for CQs specified over database schemas with KDs, in order to properly handle the presence of EDs. The rewriting is employed in the system INFOMIX [71].

Analogously to other proposals that solve consistent query answering via query rewriting (although for different classes of constraints and query languages, see, e.g., [51, 14]), the basic idea of the technique is to encode the constraints of the relational schema into a Datalog$^\neg$ program, such that the stable models of the program yield the repairs of the database instance $\mathcal{D}$. Notice that the rewriting presented below is not actually restricted to CQs, since it can be immediately extended to general Datalog$^\neg$ queries.

Definition 6.3.41 Given a CQ q and a schema $\mathcal{S}$, the Datalog$^\neg$ program $\Pi(q, \mathcal{S})$ is defined as the following set of rules. Without loss of generality, we assume that the attributes in the key precede all other attributes in r, that is $i_1 = j_1 = 1, \ldots, i_k = j_k = k$, $\ell_1 = 1, \ldots, \ell_h = h$, and $m_1 = h+1, \ldots, m_h = h+h$.

1. the rule corresponding to the definition of q;
2. for each relation $r \in \mathcal{S}$, the rules

$$\begin{aligned} r(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \quad &:\!- \quad r_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \ , \ not\ \overline{r}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \\ \overline{r}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \quad &:\!- \quad r_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \ , \ r(\vec{\mathbf{x}}, \vec{\mathbf{z}}) \ , \ y_1 \neq z_1 \\ &\ldots \\ \overline{r}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \quad &:\!- \quad r_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \ , \ r(\vec{\mathbf{x}}, \vec{\mathbf{z}}) \ , \ y_m \neq z_m \end{aligned}$$

   where: in $r(\vec{\mathbf{x}}, \vec{\mathbf{y}})$ the variables in $\vec{\mathbf{x}}$ correspond to the attributes constituting the key of the relation r; $\vec{\mathbf{y}} = y_1, \ldots, y_m$ and $\vec{\mathbf{z}} = z_1, \ldots, z_m$.
3. for each exclusion dependency $(r[i_1, \ldots, i_k] \cap\ s[j_1, \ldots, j_k]) = \emptyset$ in $\mathcal{E}$, with $r \neq s$, the rules:

$$\begin{aligned} \overline{r}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \quad &:\!- \quad r_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \ , \ s(\vec{\mathbf{x}}, \vec{\mathbf{z}}) \\ \overline{s}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \quad &:\!- \quad s_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{y}}) \ , \ r(\vec{\mathbf{x}}, \vec{\mathbf{z}}) \end{aligned}$$

   where $\vec{\mathbf{x}} = x_1, \ldots, x_k$, i.e., the variables in $\vec{\mathbf{x}}$ correspond to the sequence of attributes of r and s involved in the ED.
4. for each exclusion dependency $r[\ell_1, \ldots, \ell_h] \cap r[m_1, \ldots, m_h] = \emptyset$ in $\mathcal{E}$, the rules:

$$\begin{aligned} \overline{r}(\vec{\mathbf{x}}, \vec{\mathbf{y}}, \vec{\mathbf{z}}) \quad &:\!- \quad r_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{y}}, \vec{\mathbf{z}}) \ , \ r(\vec{\mathbf{y}}, \vec{\mathbf{w}}_1, \vec{\mathbf{w}}_2) \ , \\ \overline{r}(\vec{\mathbf{x}}, \vec{\mathbf{y}}, \vec{\mathbf{z}}) \quad &:\!- \quad r_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{y}}, \vec{\mathbf{z}}) \ , \ r(\vec{\mathbf{w}}_1, \vec{\mathbf{x}}, \vec{\mathbf{w}}_2) \ , \\ \overline{r}(\vec{\mathbf{x}}, \vec{\mathbf{x}}, \vec{\mathbf{z}}) \quad &:\!- \quad r_{\mathcal{D}}(\vec{\mathbf{x}}, \vec{\mathbf{x}}, \vec{\mathbf{z}}). \end{aligned}$$

Furthermore, we denote with $\tau(\mathcal{D})$ the database instance obtained from $\mathcal{D}$ by replacing each predicate symbol r with $r_{\mathcal{D}}$.

Informally, for each relation r, $\Pi(q, \mathcal{S})$ contains (i) a relation $r_{\mathcal{D}}$ that represents $r^{\mathcal{D}}$; (ii) a relation r that represents a subset of $r^{\mathcal{D}}$ that is consistent with the KD for r and the EDs that involve r; (iii) an auxiliary relation $\overline{r}$ that represents the "complement" of r, i.e., the subset of $r^{\mathcal{D}}$ that together with r results inconsistent with the EDs and KDs on the schema. Notice that the extension of r depends on the choice made for $\overline{r}$ (and vice-versa), and that such choices are made in a non-deterministic way (enforced by the use of the unstratified negation). The above rules force each stable model M of $\Pi(q, \mathcal{S}) \cup \tau(D)$ to be such that r^M is a maximal subset of tuples from $r^{\mathcal{D}}$ that are consistent with both the KD for r and the EDs in $\mathcal{E}$ that involve r.

Example 6.3.2 (contd.) The Datalog$^\neg$ rewriting $\Pi(q, \mathcal{S})$ of the query $q(x, z) :\!- \mathit{Journal}(x, y), \mathit{Editor}(y, z)$ is the following program:

$$
\begin{array}{rcl}
q(x,z) & :\!- & \mathit{Journal}(x,y), \mathit{Editor}(y,z) \\
\mathit{Journal}(x,y) & :\!- & \mathit{Journal}_{\mathcal{D}}(x,y)\ ,\ \mathit{not}\ \overline{\mathit{Journal}}(x,y) \\
\mathit{Editor}(x,y) & :\!- & \mathit{Editor}_{\mathcal{D}}(x,y)\ ,\ \mathit{not}\ \overline{\mathit{Editor}}(x,y) \\
\mathit{ConfPr}(x,y) & :\!- & \mathit{ConfPr}_{\mathcal{D}}(x,y)\ ,\ \mathit{not}\ \overline{\mathit{ConfPr}}(x,y) \\
\overline{\mathit{Journal}}(x,y) & :\!- & \mathit{Journal}_{\mathcal{D}}(x,y)\ ,\ \mathit{Journal}(x,z)\ ,\ z \neq y \\
\overline{\mathit{Editor}}(x,y) & :\!- & \mathit{Editor}_{\mathcal{D}}(x,y)\ ,\ \mathit{Editor}(x,z)\ ,\ z \neq y \\
\overline{\mathit{ConfPr}}(x,y) & :\!- & \mathit{ConfPr}_{\mathcal{D}}(x,y)\ ,\ \mathit{ConfPr}(x,z)\ ,\ z \neq y \\
\overline{\mathit{Journal}}(x,y) & :\!- & \mathit{Journal}_{\mathcal{D}}(x,y)\ ,\ \mathit{ConfPr}(x,z) \\
\overline{\mathit{ConfPr}}(x,y) & :\!- & \mathit{ConfPr}_{\mathcal{D}}(x,y)\ ,\ \mathit{Journal}(x,z)
\end{array}
$$

Informally, for each relation r, the above program contains (i) a relation $r_{\mathcal{D}}$ that represents $r^{\mathcal{D}}$; (ii) a relation r that represents a subset of $r^{\mathcal{D}}$ that is consistent with the KDs and the EDs for r; (iii) an auxiliary relation $\overline{r}$ that represents the "complement" of r, i.e., the subset of $r^{\mathcal{D}}$ that together with r results inconsistent with the EDs and KDs on the schema. Notice that the extension of r depends on the choice made for $\overline{r}$ (and vice-versa), and that such choices are made in a non-deterministic way (enforced by the use of the unstratified negation). In detail: the first rule of the rewriting encodes the query. The second, third and fourth rule establish the relationship between each relation and the corresponding complementary predicate. The fifth, sixth, and seventh rule encode the KDs of $\mathcal{S}$, whereas the last two rules encode the ED. ∎

We now state correctness of our encoding with respect to the semantics of consistent query answering.

Theorem 6.3.42 *let $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{E} \rangle$ be a database schema, $\mathcal{D}$ be a database instance for $\mathcal{S}$, and q be a CQ over $\mathcal{S}$. A tuple t is a consistent answer to q in $\mathcal{D}$ under $\mathcal{S}$ iff $t \in q^M$ for each stable model M of $\Pi(q, \mathcal{S}) \cup \tau(\mathcal{D})$.*

From the above theorem and Theorem 6.3.40 it follows that the consistent query answering problem under KDs and EDs is coNP-complete in data complexity.

FOL Rewriting

We now extend the first-order rewriting technique presented in previous sections to the presence of both KDs and EDs. The results presented here extends the ones of previous sections as follows:

- We refine the class $\mathcal{C}^+_{tree}$ in order to obtain a class of queries, called $\mathcal{KE}$*-simple*, for which consistent query answering is polynomial in the presence of both KDs and also EDs.
- We provide a new algorithm for computing the FOL rewriting for $\mathcal{KE}$*-simple* queries.

Let us describe in detail our technique. Henceforth, given a CQ q, we denote by R_q the set of relation symbols occurring in $body(q)$. Given a database schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{E} \rangle$ and a CQ q, we denote by $O_{\mathcal{E}}(q)$ the set of relation symbols $O_{\mathcal{E}}(q) = \{s \mid r[j_1, \ldots, j_k] \cap s[\ell_1, \ldots, \ell_k] = \emptyset \in \mathcal{E} \text{ and } r \in R_q\}$. In words, $O_{\mathcal{E}}(q)$ contains each relation symbol $s \in \mathcal{A}$ such that there exists an exclusion dependency between s and r in $\mathcal{E}$, where r is a relation symbol occurring in $body(q)$.

Definition 6.3.43 Let $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{E} \rangle$ be a database schema. A conjunctive query q is $\mathcal{KE}$*-simple* if $q \in \mathcal{C}^+_{tree}$, and

- there exists no pair of relation symbols r, s in $O_{\mathcal{E}}(q)$ such that there exists an exclusion dependency between r and s in $\mathcal{E}$,
- there exists no relation symbol r in $O_{\mathcal{E}}(q)$ such that there exists $r[i_1, \ldots, i_k] \cap s[j_1, \ldots, j_k] = \emptyset$ in $\mathcal{E}$, and either $key(r) \not\supseteq \{i_1, \ldots, i_k\}$ or $key(s) \not\supseteq \{j_1, \ldots, j_k\}$, where s is a relation symbol in R_q.

In words, a query q is $\mathcal{KE}$-simple if it belongs to the class $\mathcal{C}^+_{tree}$, and if both there are no EDs between relations that are in $O_{\mathcal{E}}(q)$, and each ED between a relation $r \in R_q$ and a relation $s \in O_{\mathcal{E}}(q)$ does not involve non-key attributes of r or s. Notice that this last condition does not limit the applicability of our approach in many practical cases. For example, in relational databases obtained from ER-schemas, EDs are typically specified between keys.

For $\mathcal{KE}$-simple CQs, we present in the following a query rewriting algorithm which, given a query q, produces a FOL rewriting, whose evaluation over any database instance $\mathcal{D}$ for the database schema $\mathcal{S}$ returns the consistent answers to q in $\mathcal{D}$ under $\mathcal{S}$. The basic idea of the algorithm is to specify a set of conditions, expressible in FOL, that, if verified over a database instance $\mathcal{D}$, for a given tuple t, guarantee that in any repair of $\mathcal{D}$ there is an image of t w.r.t q, i.e., t is a consistent answer to q in $\mathcal{D}$. We point out that, for non-$\mathcal{KE}$-simple CQs, such conditions cannot be specified in FOL.

The following algorithm CQFOLReduction computes the FOL rewriting to a $\mathcal{KE}$-simple conjunctive query q. In the algorithm, $JG(q)$ denotes the typed join graph of q defined in 6.3.9 (notice that for $\mathcal{KE}$-simple queries the join graph is a forest, since it is acyclic).

Algorithm CQFOLReduction$(q, \mathcal{S})$
Input: $\mathcal{KE}$-simple CQ q (whose head variables are $x_1, \ldots, x_n$);
schema $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{E} \rangle$

Output: FOL query (representing the rewriting of q)
begin
compute $JG(q)$;
return $\{x_1, \ldots, x_n \mid \bigwedge_{N \in roots(JG_{(q)})} \mathsf{FolTree}(N, \mathcal{E})\}$
end

Basically, the algorithm builds the join graph of q and then builds the first-order query by invoking the algorithm FolTree on all the nodes that are roots of the join graph. The algorithm FolTree is defined in Figure 6.11.

Roughly speaking, the algorithm $\mathsf{FolTree}(N, \mathcal{E})$ returns a first-order formula that constitutes the encoding of the whole subtree of the join graph of the query whose root is the node N. To do that, the algorithm computes two sub-formulas f_1 and f_2.

The formula f_1 contains an atom whose predicate is the predicate r labelling the node N, in which the unbound variables of r are renamed with new existentially quantified variables. Furthermore, f_1 contains an atom of the form $\neg\exists y'_{i_1}, \ldots, y'_{i_k} . s(z_1, \ldots, z_m)$ for each ED that involves r and a relation s. Intuitively, when evaluated over a database instance $\mathcal{D}$, each such atom checks that there are no facts of the form $s(t_s) \in \mathcal{D}$ that violate the ED together with a fact of the form $r(t_r) \in \mathcal{D}$, which is in an image I of a tuple t w.r.t. the input query q, i.e., the atom guarantees that I is not contradicted w.r.t. the ED.

The formula f_2 is empty only when all non-key arguments of the atom r are existential non-shared variables (i.e., of type U). Otherwise, the formula f_2 is a universally quantified implication. In such an implication, the antecedent is an atom whose predicate is r, and the consequent is a conjunction of equality conditions and other sub-formulas: more precisely, there is an equality condition for each non-key argument in r of type B, and a sub-formula for each successor N' of N in the join graph of q, computed by recursively invoking FolTree on N'.

Intuitively, f_2 enforces the joins between r and each atom labelling the successors of r in the join graph of q. At the same time f_2 ensures that, when evaluated over a database instance $\mathcal{D}$, if there exists a fact of the form $r(\overline{t_r}) \in \mathcal{D}$ that violates the KD specified on r together with a fact of the form $r(t_r) \in \mathcal{D}$, which is in the image of a tuple t w.r.t. q, $r(\overline{t_r})$ belongs to another image of t w.r.t. q. In other words, the atom guarantees that in any repair there exists an image of t (w.r.t. the KD on r). Such a check is iterated for other KDs by recursively invoking FolTree. The following example illustrates the way the algorithm works.

Example 6.3.2 (contd.) It is easy to verify that the query $q(x, z) :\!- Journal(x, y), Editor(y, z)$ is $\mathcal{KE}$-simple. The join graph of such a query is depicted in Figure 6.12.

Now, by applying the algorithm CQFOLReduction and FolTree we obtain:

$$\begin{aligned}
\mathsf{CQFOLReduction}(q) &= \{x, z \mid \mathsf{FolTree}(N1)\} \\
\mathsf{FolTree}(N1) &= \exists y_2.\ Journal(x, y_2) \wedge \neg\exists y'_2.\ ConfPr(x, y'_2) \wedge \\
&\quad \forall y.\ Journal(x, y) \rightarrow (\mathsf{FolTree}(N2)) \\
\mathsf{FolTree}(N2) &= Editor(y, z) \wedge \forall y''_2.\ Editor(y, y''_2) \rightarrow y''_2 = z.
\end{aligned}$$

Algorithm FolTree(N,$\mathcal{E}$)
Input: node N of $JG(q)$; set of EDs $\mathcal{E}$
Output: FOL formula
begin
let $a = r(x_1/t_1, \ldots, x_n/t_n)$ be the label of N;
for $i := 1$ **to** n **do**
if $t_i \in \{KB, B\}$ **then** $v_i := x_i$
else $v_i := y_i$, where y_i is a new variable
if each argument of a is of type B or KB **then** $f_1 := r(x_1, \ldots, x_n)$
else begin
let $i_1, \ldots, i_m$ be the positions of the arguments of a of type S, U, KU;
$f_1 := \exists y_{i_1}, \ldots, y_{i_m}.\, r(v_1, \ldots, v_n)$
end;
for each ED $r[j_1, \ldots, j_k] \cap s[\ell_1, \ldots, \ell_k] = \emptyset \in \mathcal{E}$ **do**
begin
let m be the arity of s;
for $i := 1$ **to** m **do**
if $i \in \{\ell_1, \ldots, \ell_k\}$ **then if** $i = \ell_c$ **then** $z_i = v_{j_c}$
else $z_i = y'_i$ where y'_i is a new variable;
let $y'_{i_1}, \ldots, y'_{i_k}$ be the new variables above introduced;
$f_1 = f_1 \wedge \neg\exists y'_{i_1}, \ldots, y'_{i_k}\, .\, s(z_1, \ldots, z_m)$
end
if there exists no argument in a of type B or S **then return** f_1
else begin
let $p_1, \ldots, p_c$ be the positions of the arguments of a of type U, S or B;
let $\ell_1, \ldots, \ell_h$ be the positions of the arguments of a of type B;
for $i := 1$ **to** c **do**
if $t_{p_i} = S$ **then** $z_{p_i} := x_{p_i}$ **else** $z_{p_i} := y''_i$, where y''_i is a new variable
for $i := 1$ **to** n **do**
if $t_i \in \{KB, KU\}$ **then** $w_i := v_i$ **else** $w_i := z_i$;
$$f_2 := \forall z_{p_1}, \ldots, z_{p_c}.\, r(w_1, \ldots, w_n) \rightarrow \left(\bigwedge_{N' \in jgsucc(N)} \mathsf{FolTree}(N') \right) \wedge \bigwedge_{i \in \{\ell_1, \ldots, \ell_h\}} w_i = x_i$$
return $f_1 \wedge f_2$
end
end

Figure 6.11: The algorithm FolTree

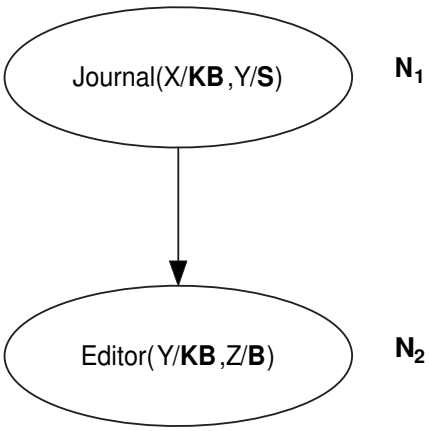


Figure 6.12: Typed join graph for the query of Example 6.3.2

By evaluating the rewriting over $\mathcal{D}$ we get $\{\langle \mathsf{TODS}, \mathsf{USA} \rangle\}$, i.e., the set of consistent answers to q in $\mathcal{D}$ under $\mathcal{S}$. □

■

Next, we state soundness and completeness of the algorithm.

Theorem 6.3.44 *Let $\mathcal{S} = \langle \mathcal{A}, \mathcal{K}, \mathcal{E} \rangle$ be a database schema, q be a $\mathcal{KE}$-simple conjunctive query over $\mathcal{S}$, and q_r be the FOL rewriting returned by* CQFOLReduction*(q). Then, for every database instance $\mathcal{D}$ for $\mathcal{S}$, a tuple t is a consistent answer to q in $\mathcal{D}$ under $\mathcal{S}$ iff $t \in q_r^{\mathcal{D}}$.*

As a corollary, consistent query answering for $\mathcal{KE}$-simple conjunctive queries over database schemas with KDs and EDs is polynomial in data complexity.

6.4 Efficient Data Integration under Integrity Constraints

In the previous section we presented first-order reducibility of CQA of (union of) conjunctive queries for a variety of integrity constraints. The advantage of such kind of rewriting is twofold: on one hand, first-order queries can be evaluated efficiently over inconsistent instances of a database in order to retrieve the consistent answers; on the other hand, first-order queries can be easily translated into SQL queries and thus evaluated using standard database technologies.

In order to apply the first-order rewriting technique in a data integration setting, we generalize the loosely-sound semantics presented in Section 6.3.1 to such more complex setting. To this aim we define an ordering relation $\leq_{\mathcal{D}}$ on the set of all global database $\mathcal{B}$ for $\mathcal{G}$, such that given two global databases $\mathcal{B}_1$ and $\mathcal{B}_2$, $\mathcal{B}_1 \leq_{\mathcal{D}} \mathcal{B}_2$ if for every mapping assertion $\langle g, q_{\mathcal{S}} \rangle \in \mathcal{M}$[2]: ${g^{\mathcal{B}_1} \cap q_{\mathcal{S}}}^{\mathcal{D}} \supseteq {g^{\mathcal{B}_2} \cap q_{\mathcal{S}}}^{\mathcal{D}}$. Moreover, given two databases $\mathcal{B}_1$ and $\mathcal{B}_2$, $\mathcal{B}_1 <_{\mathcal{D}} \mathcal{B}_2$ if $\mathcal{B}_1 \leq_{\mathcal{D}} \mathcal{B}_2$ and $\mathcal{B}_2 \not\leq_{\mathcal{D}} \mathcal{B}_1$. By using the ordering relation above, we define the loosely-sound semantics for a data integration system $\mathcal{I}$, denoted by $sem_{ls}(\mathcal{I}, \mathcal{D})$, as the set of global database $\mathcal{B}$ such that:

[2]We consider here only GAV mappings under the sound assumption

- $\mathcal{B}$ is consistent with $\mathcal{G}$
- $\mathcal{B}$ is minimal with respect to $\leq_{\mathcal{D}}$, i.e., no other global databases $\mathcal{B}'$ consistent with $\mathcal{G}$ exist such that $\mathcal{B}' <_D \mathcal{B}$.

The notions of perfect rewriting and answers to a user query q coincide with the ones proposed in Chapter 2 for the strict semantics using $sem_{las(\mathcal{M})}(\mathcal{I}, \mathcal{D})$ instead of $sem_{as(\mathcal{M})}(\mathcal{I}, \mathcal{D})$.

Example 6.4.1 Let us consider the data integration system $\mathcal{I}'$ obtained from the one presented in Example 2.2.1 by adding the key constraint $key(\mathsf{musician}) = \{1\}$ to the set of integrity constraints Σ of $\mathcal{G}$. It should be easy to see that with this new constraint also $sem_s(\mathcal{I}, \mathcal{D}) = \emptyset$: every global database must contain the facts $\mathsf{musician}(luckater, TOTO)$ and $\mathsf{musician}(luckater, U2)$ introducing an inconsistency with the respect of the key dependency that can not be repaired by adding tuples to the global databases.

Contrarily, with regard to the loose semantics presented above, we are now able to add as well as remove tuples from the global databases, obtaining:

- under the *loosely-sound* semantics two global databases exist of the form:

$$\begin{aligned}\mathcal{B}_1 &= \{\mathsf{musician}(luckater, TOTO), \mathsf{musician}(townsend, WHO),\\ &\quad \mathsf{band}(TOTO, USA), \mathsf{band}(WHO, \alpha)\}\\ \mathcal{B}_2 &= \{\mathsf{musician}(luckater, U2), \mathsf{musician}(townsend, WHO),\\ &\quad \mathsf{band}(TOTO, USA), \mathsf{band}(WHO, \alpha), \mathsf{band}(U2, \beta)\}\end{aligned}$$

  in which α and β are constants of the domain, plus others global databases obtaining from $\mathcal{B}_1$ and $\mathcal{B}_2$ by adding facts to them. Under this semantics the answers to the query $q(X) \leftarrow \mathsf{band}(X, Y)$ are $ans_{l_s}(q, \mathcal{I}, \mathcal{D}) = \{TOTO, WHO\}$. $\mathcal{U}$.

- under the *loosely-exact* semantics the possible global databases are $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4$, where:

$$\begin{aligned}\mathcal{B}_3 &= \{\mathsf{musician}(luckater, TOTO), \mathsf{band}(TOTO, USA)\}\\ \mathcal{B}_4 &= \{\mathsf{musician}(luckater, U2), \mathsf{band}(TOTO, USA),\\ &\quad \mathsf{band}(U2, \alpha)\}\end{aligned}$$

  in which α is a constant of the domain $\mathcal{U}$. For the query $q(X) \leftarrow \mathsf{band}(X, Y)$ we have that $ans_{l_e}(q, \mathcal{I}, \mathcal{D}) = \{TOTO\}$

As final remark, notice that under the complete assumption, the loose semantics coincide with the strict semantics and $ans_c(q, \mathcal{I}, \mathcal{D}) = ans_{l_c}(q, \mathcal{I}, \mathcal{D}) = \emptyset$. ∎

Let us turn our attention on the query processing task: we extend the system presented in Chapter 5 by providing it with a new module that is responsible for the treatment of integrity constraints. In detail the new module, called *Integrity Constraints Processor*, takes the user query as input, and produces a first order rewriting of it by means of the algorithms presented in the previous section. Such FOL query is then translated into SQL and sent to the Query Reformulator (see Section 5.2.5 and Section 5.3 for details) that is responsible for the treatment

of such reformulated query, according to the internal technique presented in Section 5.3.1. In words, the Query Processor module rewrite the FOL query produced by the Integrity Constraint Processor, and replace each relation symbol with the name of the corresponding view created according to the mapping of the system specification.

Example 6.4.2 Consider again the data integration system specification presented in Example 5.3.1, and assume that the first attributes of the global relation owner and building are key attributes for those relations. Let us consider now the following query:

$$\mathsf{q}(X, Z) \quad :- \quad \mathsf{building}(X, Y), \mathsf{owner}(Y, Z).$$

The first order rewriting of such query according to the CQRewrite algorithm is the following:

$$\{X, Z \mid \exists y_2.\ \mathsf{building}(X, y_2) \wedge \forall y.\ \mathsf{building}(Z, y) \rightarrow (\mathsf{owner}(y, Z) \wedge \forall y_2''.\ \mathsf{owner}(y, y_2'') \rightarrow y_2'' = Z)\}$$

Such FOL query is then processed by the Query Reformulator according to the internal technique: the output of this step is an analogous first order formula, in which owner and Building are replaced with V_{owner} and V_{building}. Such new query is then expressed in SQL and evaluated over the compiled data integration system specification produced by the Mapping Processor. ∎

The new architecture of the system is presented in Figure 6.13. The correctness of the whole approach is ensured by well-known results on the unfolding technique, and by the theorem shown in the previous Section.

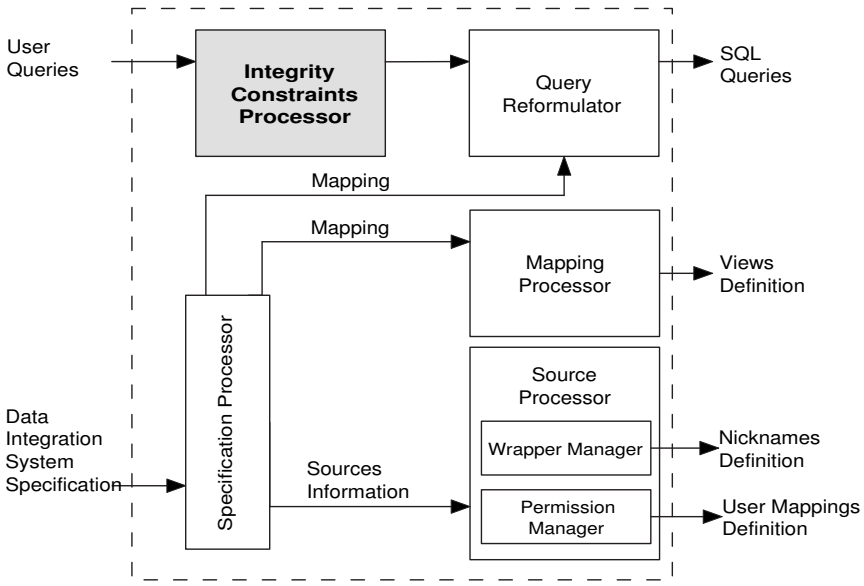


Figure 6.13: The new architecture of the system

Chapter 7

Experiments

In this section we present the results of several experiments we have conducted to test the performance of our prototype. The experiments were conducted on a real large test database whose schema and size are presented in Section 7.1. Section 7.2 presents a comparison between the two different techniques for handling mappings described in Chapter 5. In Section 7.3 we test the efficiency of the first-order rewriting for querying inconsistent databases, comparing such an approach to the one based on rewriting the query in a more complex logic program. Finally, in Section 7.4 we compare Oracle 10g, IBM DB2 Information Integrator and Microsoft SQL Server 2005 from the point of view of sources access.

7.1 Experiment Scenario

We conducted all the experiments presented in this chapter on a real large database of the University of Rome "La Sapienza". The database holds career information about students of the Computer Science Engineering degree course and bachelor course, as well as teaching information of the whole athenaeum, within 27 different tables. The relational schema of the database is shown in Figure 7.1. In turn, Figure 7.2 shows the number of tuples for each table in the database. The size of the whole database is around 250.000 tuples.

The experiments were conducted on a double processor machine, with 3 GHz Pentium IV Xeon CPU and 2 GB of main memory, running both the Linux and Windows 2003 Server operating systems. We executed a set of significant queries over the database. We ran every query 5 times through a JDBC connection, reporting in the charts the average time of the 5 executions in order to avoid any kind of system interference. Notice that in all the experiments, we always disregarded the query rewriting time (when a rewriting has been produced): indeed, usually it does not affect significantly the whole query execution time.

bachelor_exam	(CODE,DESCRIPTION)
bachelor_exam_record	(STUDENT_ID,LAST_NAME,FIRST_NAME,EXAM_ID, PROFESSOR,SESSION,CALL,YEAR,MODALITY, RATING,LAUD,ACCADEMIC_YEAR)
career	(STUDENT_ID,ACADEMIC_YEAR,COURSE_YEAR, ENROLLMENT_TYPE,FACULTY,DEGREE_COURSE, UNIVERSITY,CAREER_STATUS,YEAR_CAREER_VALIDITY, INCOME_BAND,FAMILY_MEMBERS)
career_status	(CODE,DESCRIPTION)
course	(FACULTY_CODE,CODE,DESCRIPTION)
course_assignment	(EXAM_CODE,PROFESSOR_CODE,ACCADEMIC_YEAR)
course_exam	(COURSE_FACULTY,COURSE_CODE,EXAM_FACULTY, EXAM_CODE)
degree	(STUDENT_ID,THESIS_TITLE,THESIS_DATE,RATING,ADVISOR)
degree_course	(UNIVERSITY,FACULTY,CODE,DESCRIPTION)
enrollment	(CODE,DESCRIPTION)
exam	(FACULTY_CODE,CODE,DESCRIPTION,ACTIVATION)
exam_plan	(CODE,STUDENT_ID,POSITIONING,PRESENTATION_DATE, STATUS,NOTES,PROP_RESP,BASE,ADDRESS_A,ADDRESS_B)
exam_plan_data	(CODE,EXAM_CODE,NAME)
exam_plan_status	(CODE,DESCRIPTION)
exam_regularity	(CODE,DESCRIPTION)
exam_type(	CODE,DESCRIPTION)
faculty	(UNIVERSITY_CODE,CODE,DESCRIPTION)
high_school	(CODE,DESCRIPTION)
master_exam	(CODE,DESCRIPTION,TYPE,EXAM_YEAR)
master_exam_record	(STUDENT_ID,COURSE_CODE,EXAM_CODE, EXAM_DATE,RATING,REGULARITY,ACADEMIC_YEAR)
modality	(CODE,DESCRIPTION)
positioning	(CODE,DESCRIPTION)
professor	(CODE,FIRST_NAME,LAST_NAME,MATTER)
professor_data	(CODE,FIRST_NAME,LAST_NAME)
session	(CODE,DESCRIPTION)
student	(STUDENT_ID,LAST_NAME,FIRST_NAME,BIRTH_DAY, BIRTH_PLACE,BIRTH_CONTRY,DOMICILE_ADDRESS, DOMICILE_STREET_NUMBER,DOMICILE_POSTAL_CODE, DOMICILE_CITY,DOMICILE_COUNTRY,DOMICILE_PREFIX, DOMICILE_PHONE,RESIDENCE_ADDRESS, RESIDENCE_STREET_NUMBER,RESIDENCE_POSTAL_CODE, RESIDENCE_CITY,RESIDENCE_COUNTRY, RESIDENCE_PREFIX,RESIDENCE_PHONE,SSN, HIGH_SCHOOL_TYPE,HIGH_SCHOOL_RATING)
university	(CODE,CITY,DESCRIPTION)

Figure 7.1: Relational schema of the test database

Table	Size	Table	Size
bachelor_exam	28	exam_regularity	4
bachelor_exam_record	17001	exam_type	3
career	50633	faculty	511
career_status	15	high_school	69
course	4722	master_exam	66
course_assignment	402	master_exam_record	106268
course_exam	7204	modality	2
degree	397	positioning	29
degree_course	1716	professor	146
enrollment	5	professor_data	67
exam	17144	session	4
exam_plan	1089	student	16079
exam_plan_data	27130	university	163
exam_plan_status	3		

Figure 7.2: Tables size (number of tuples)

7.2 Comparing techniques for handling mapping

In this section we compare the two rewriting techniques presented in Chapter 5 in order to process the mappings of a GAV data integration system. To this aim, we built one instance of the test database on Oracle 10g, one instance of IBM DB2 and one instance of SQL Server 2005, using each such instance as a set of sources for a data integration system. We then defined a global schema and a set of GAV mappings over those sources. The specification of the whole system contains the following relational global schema:

$$
\begin{aligned}
g\_student \quad & (ID, FIRST\_NAME, SECOND\_NAME, \\
& CITY\_OF\_RESIDENCE, ADDRESS, TELEPHONE, \\
& HIGH\_SCHOOL\_SPECIALIZATION) \\
g\_enrollment \quad & (STUDENT\_ID, FACULTY\_NAME, YEAR) \\
g\_exam \quad & (CODE, DESCRIPTION) \\
g\_professor \quad & (CODE, FIRST\_NAME, SECOND\_NAME) \\
g\_university \quad & (U\_CODE, CITY, NAME) \\
g\_master\_exr \quad & (STUDENT, EXAM, PROF\_CODE, MARK, \\
& DDATE, COURSE\_YEAR) \\
g\_bachelor\_exr \quad & (STUDENT, EXAM, SSESSION, MMODE, MARK) \\
g\_teaching \quad & (COURSE, PROFESSOR, ACADEMIC\_YEAR) \\
g\_exam\_plan \quad & (CODE, STUDENT, PLAN\_TYPE, \\
& REQUEST\_DATE, STATUS) \\
g\_exam\_plan\_data \quad & (CODE, EXAM\_NAME, EXAM\_TYPE)
\end{aligned}
$$

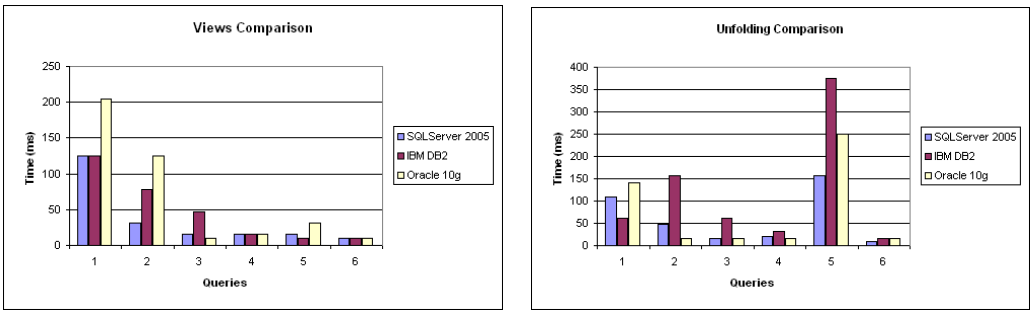
$$
\begin{aligned}
g\_student(X_1, X_2, X_3, X_4, X_5, X_6, X_7) \quad &:- \quad student(X_1, X_3, X_2, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, \\
&\qquad A_9, X_6, X_5, A_{10}, A_{11}, X_4, A_{12}, A_{13}, A_{14}, A_{15}, Y, A_{16}), \\
&\qquad high\_school(Y, X_7). \\
g\_enrollment(X_1, X_2, X_3) \quad &:- \quad career(X_1, X_3, A_1, A_2, Y, A_3, A_4, A_5, A_6, A_7, A_8), \\
&\qquad faculty(A_9, Y, X_2). \\
g\_exam(X_1, X_2) \quad &:- \quad exam(A_1, X_1, X_2, A_2). \\
g\_university(X_1, X_2, X_3) \quad &:- \quad university(X_1, X_2, X_3). \\
g\_professor(X_1, X_2, X_3) \quad &:- \quad professor(X_1, X_2, X_3, A_1). \\
g\_professor(X_1, X_2, X_3) \quad &:- \quad professor\_data(X_1, X_2, X_3). \\
g\_master\_exr(X_1, X_2, X_3, X_4, X_5, Y) \quad &:- \quad master\_exam\_record(X_1, A_1, X_2, X_5, X_4, A_2, Y), \\
&\qquad course\_assignment(X_2, X_3, Y). \\
g\_teaching(X_1, X_2, X_3) \quad &:- \quad course\_assignment(X_1, X_2, X_3). \\
g\_bachelor\_exr(X_1, X_2, X_3, X_4, X_5) \quad &:- \quad bachelor\_exam\_record(X_1, A_1, A_2, X2, A_3, Y_1, \\
&\qquad A_4, A_5, Y_2, X_5, A_6, A_7), \\
&\qquad modality(Y2, X_4), session(Y1, X_3). \\
g\_exam\_plan(X_1, X_2, X_3, X_4, X_5) \quad &:- \quad exam\_plan(X_1, X_2, Y_1, X_4, Y_2, A_1, A_2, A_3, A_4, A_5), \\
&\qquad positioning(Y_1, X_3), exam\_plan\_status(Y_2, X_5). \\
g\_exam\_plan\_data(X_1, X_2, X_3) \quad &:- \quad exam\_plan\_data(X_1, Y1, A_1), \\
&\qquad master\_exam(Y1, X_2, Y2, A_2), \\
&\qquad exam\_type(Y2, X_3).
\end{aligned}
$$

Figure 7.3: GAV mappings for the test data integration system

The source schema is identical to the relational schema of the test database presented in Figure 7.1, while the GAV mapping specified between the source schema and the global schema above is presented in Figure 7.3. For easy of prentation, we indicated head variables with the "X" symbol, existential join variable with the "Y" symbol and existential non-join variables with the "A" symbol, each symbol possibly with subscript. We tested the following queries expressed over the global schema of the data integration system instance presented above.

$$
\begin{aligned}
Q_1(X_1, X_2, X_3) &:- g\_student(Y_1, A_2, X_1, A_4, A_5, X_3, A_7),\\
&\quad g\_master\_exr(Y_1, Y_2, B_3, B_4, B_5, B_6),\\
&\quad g\_exam(Y_2, X_2).\\
Q_2(X_1, X_2) &:- g\_master\_exr(A_1, Y_1, Y_2, 30, A_5, A_6),\\
&\quad g\_professor(Y_2, A_7, X_2),\\
&\quad g\_exam(Y_1, X_1).\\
Q_3(X_1, X_2, X_3) &:- g\_student(Y_1, X_2, X_3, A_4, A_5, A_6, A_7),\\
&\quad g\_enrollment(Y_1, X_1, B_3, 1998).\\
Q_4(X_1, X_2) &:- g\_exam\_plan(Y_1, A_2, A_3, A_4, A_5),\\
&\quad g\_exam\_plan\_data(Y_1, X_1, X_2).\\
Q_5(X_1, X_2, X_3) &:- g\_teaching(X_1, Y_1, X_3),\\
&\quad g\_professor(Y_1, A_1, X_2).\\
Q_6(X_1, X_2) &:- g\_student(A_1, A_2, X_1, X_2, A_5, A_6, A_7).
\end{aligned}
$$

We used both the *internal* and the *external* technique for handling GAV mappings (see Section 5.3 for details). As for the internal technique, the *Mapping Processor* module (see Section 5.2.4) defines a view for every global relation, based on the corresponding GAV mapping assertion. Conversely, according to the external technique, the mappings are processed at query time, translating the query by means of an unfolding algorithm implemented by the *Query Reformulator* module (see Section 5.2.5). Queries produced by this module are expressed over the relations of the source schema and evaluated straightforwardly over the source database instance. We compare execution times for both the techniques in Figure 7.4 and 7.5. In the first one we report execution times of the internal technique (labelled with "view comparison" in the charts) aver the three DBMSs (Figure 7.4(a)) and the execution time of the external technique (labelled "unfolding" in the charts) over such DBMSs (Figure 7.4(b)). In Figure 7.5 we reorganize such results and show execution times of both the techniques separately for each DBMS.



(a) Internal technique execution time (b) External technique execution time

Figure 7.4: Experimental Results

As it can be seen in Figure 7.5(a), for IBM DB2 Information Integrator the internal technique performs always better than the external one (unfolding) except for the query Q_1. For the Oracle 10g system (Figure 7.5(b)) the internal techniques shows better performance than the external one only for queries Q_3, Q_5 and Q_6, while SQL Server 2005 behaves substantially like IBM DB2

except for queries Q_3 and Q_6 for which execution times of the internal and external techniques are almost the same (Figure 7.5(c)). Furthermore, Figure 7.5 shows that SQL Server 2005 is on the average faster than Oracle 10g and IBM DB2, for both the internal and the external technique.

Therefore, these experimental results show that there is no way of establishing that one technique must be preferred to the other. This may be surprising because one might expect that the SQL engine that processes the views into a DBMS is more efficient than an unfolding algorithm. however, the reason of the different behaviors of the different systems with respect to different queries is mainly due to how the SQL engines of the three systems treat and optimize the views that are employed in the internal technique to represent the GAV mappings: we point ut that, a deeper analysis of such an issue is out of the scope of this thesis.

7.3 Experimenting the efficiency of FOL rewritings for CQA

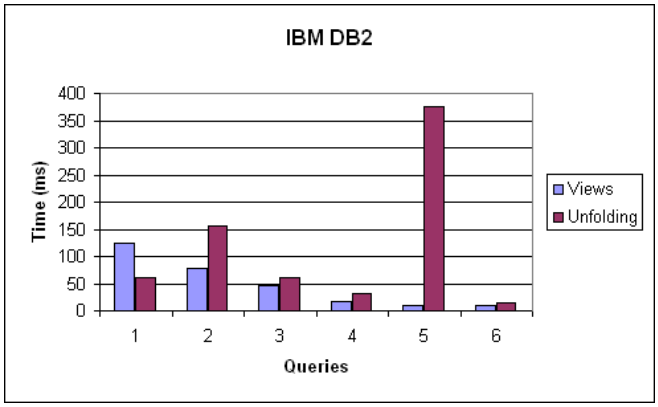
We now present some experimental results comparing the FOL and the Datalog$^\neg$ rewriting described in Section 6.3.6. For the aim of such investigation we focus here on the setting of a single database. In particular we use for our experiments the test databases described in Section 7.1, enriched with the following set of integrity constraints.

relations		**integrity constraints**	
faculty	*exam_plan*	$key(faculty) = \{1, 2\}$	$key(plan\_status) = \{1\}$
course_assignment	*degree*	$key(exam\_plan) = \{1\}$	$key(positioning) = \{1\}$
positioning	*course*	$key(university) = \{1\}$	$key(prof\_data) = \{1\}$
plan_status		$key(exam\_type) = \{1\}$	$key(degree) = \{1\}$
prof_data		$key(course) = \{1\}$	$key(exam) = \{2\}$
university		$key(master\_exam) = \{1\}$	
bachelor_exam		$key(bachelor\_exam) = \{1\}$	
master_exam		$course\_assignment[2] \cap professor[1] = \emptyset$	
exam_type		$master\_exam[1] \cap bachelor\_exam[1] = \emptyset$	
exam		$course[3, 4] \cap bachelor\_exam[1, 2] = \emptyset$	

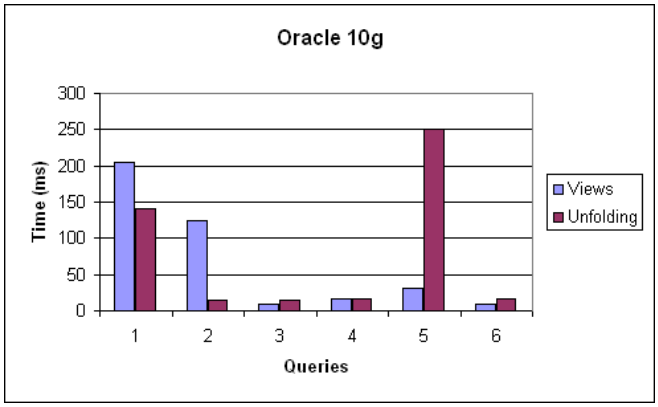
To perform the experiments, we implemented a rewriting module that operates according to the CQFOLReduction algorithm presented in Section 6.3.6, which translates CQs issued over the database schema into FOL queries. Such queries are then encoded by the rewriting module into SQL.

Another software component, specifically developed for testing purposes, and that is not part of the system, is responsible for the generation of the Datalog$^\neg$ programs according to Definition 6.3.41. Such programs are then evaluated by the stable model engine DLV [72] over the database instance.

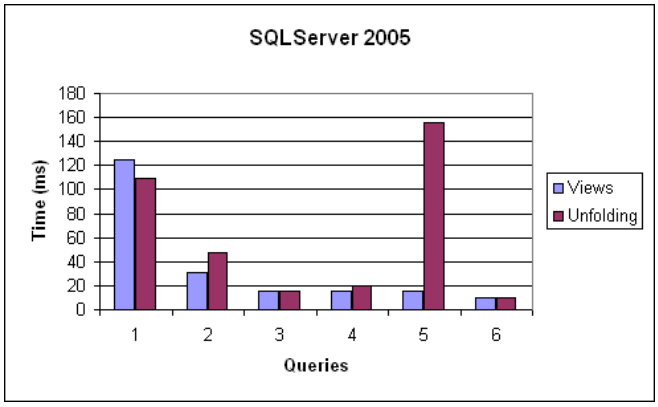
As for the DBMS that stores our database, we chose the MySQL database system instead of one of the three systems presented in Chapter 4. Such a choice is motivated by the necessity to test the feasibility of FOL rewriting also on light-weight DBMSs. Moreover, we chose DLV since



(a) Internal vs External technique for IBM DB2



(b) Internal vs External technique for Oracle 10g



(c) Internal vs External technique for SQL Server 2005

Figure 7.5: Experimental Results

it is a state-of-the-art reasoning engine for Datalog$^\neg$ programs.

We executed ten queries over the database. Among such queries we selected the following three queries, that are the most significant:

$$\begin{aligned} Q_0(C) &:- faculty(C, U, 'INGEGNERIA'). \\ Q_2(S, D, P) &:- positioning(PS, P), plan\_status(ST, D), \\ &\quad exam\_plan(C, S, PS, DT, ST, '1', U1, U2, U3, U4). \\ Q_3(N, D, NP, CP) &:- master\_exam(C, N, T, '5'), \\ &\quad exam\_type(T, D), course\_assignment(C, P, A), \\ &\quad professor\_data(P, NP, CP). \end{aligned}$$

whose execution times are shown in Figure 7.6.

The queries have been posed on various instances of the test database with an increasing number of pairs of tuples violating some ICs.

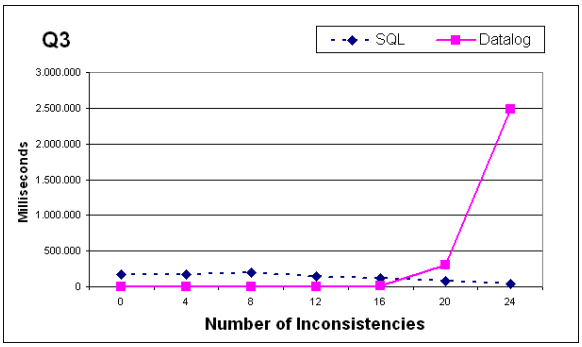
In charts 7.6(a), 7.6(b) and 7.6(c), the execution times of the SQL encoding and of the Datalog$^\neg$ program are compared for queries Q_0, Q_2, and Q_3 respectively. As expected, from a certain inconsistency level on, the execution time of the Datalog$^\neg$ encoding has an exponential blow-up; in contrast, the execution time for the SQL encoding is constant on the average, and for Q_3 (Figure 7.6(b)) it decreases: although this might be surprising, it turns out that some inconsistency allows the SQL engine to prune the search space for query answering.

Furthermore, the chart presented in Figure 7.6(d) compares, on a logarithmic scale, the execution time of all queries at the highest inconsistency level. It shows that the SQL encoding is always more efficient when the degree of data inconsistency is high. Notice that the method based on Datalog$^\neg$ and DLV is particularly efficient in the presence of few data inconsistencies.

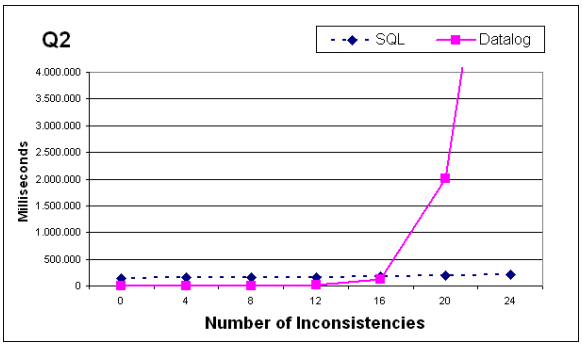
7.4 Testing source access

We finally turn back to a data integration system and analyze the way in which the three commercial DBMSs used for constructing our system access the sources. In order to compare Oracle 10g, IBM DB2 and SQL Server 2005, we built some non-relational data sources starting from the test database instance. In particular we produced an Excel workbook containing worksheets whose size ranges from 28 to 17144 lines. We also produced several XML documents based on tables up to 7000 tuples. We then wrapped Excel and XML sources on every commercial tool and tested it with some test queries. In particular we issued `select *` and `select count(*)` queries without selections on every data source comparing the obtained execution times in Figure 7.7. As an immediate result, it is easy to see that SQL Server 2005 is always the fastest system: this is due to the fact that SQL Server 2005 physically imports data sources into local tables, while Oracle 10g and IBM DB2 access sources at query time. A further interesting result is that for Excel sources the size of the worksheet does not affect the query time, while for XML documents it strongly grows with the size of the document (notice that the charts on Figure 7.7(b) and Figure 7.7(d) are in a logarithmic scale).

(a) Q_0 execution time



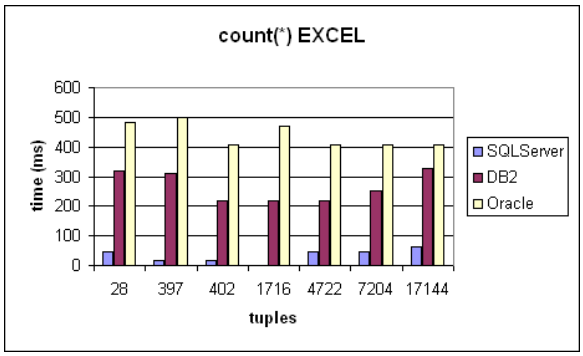
(b) Q_3 execution time



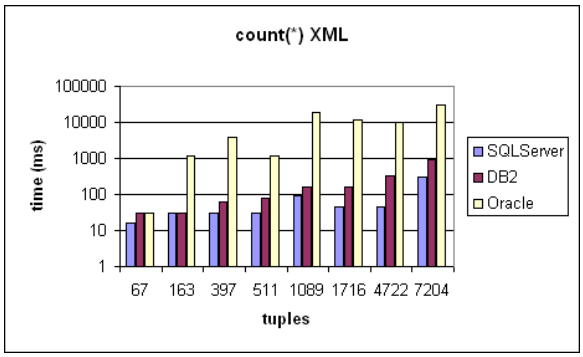
(c) Q_2 execution time

(d) SQL vs. Datalog¬

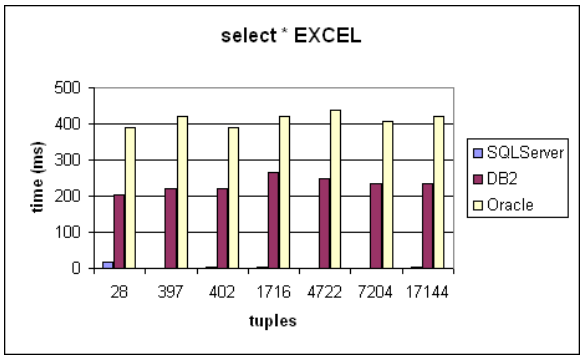
Figure 7.6: Experimental Results



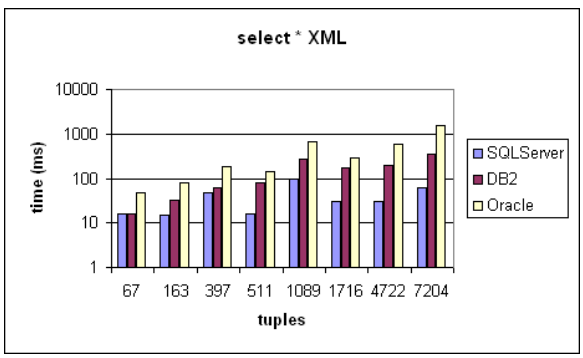
(a) `count(*)` query on Excel data sources



(b) `count(*)` query on XML data sources



(c) `select *` query on Excel data sources



(d) `select *` query on XML data sources

Figure 7.7: Experimental Results

Chapter 8

Conclusions

In this thesis we have presented an approach to the construction of efficient data integration systems. First of all, we have analyzed three top-line commercial data integration tools: Oracle 10g Information Integration, IBM DB2 Information Integrator and SQL Server 2005 Integration Service. Then, starting from some limitations enforced by all such systems in order to handle real practical integration cases, we have presented a novel system aiming at filling the gap between the real needs of the data integration issues and the features provided by commercial tools. In particular, the system presented allows the users to build an expressive global schema, representing the users' domain of interest, and to connect such schema with several heterogeneous sources through a GAV mapping. Two main techniques have been implemented within the system in order to process such a mapping: one that exploits the capability of the underlying commercial tools to efficiently treat views, and another one that handle the mapping by means of a rewriting module. We point out that the possibility of defining an expressive global schema, and in particular the possibility of expressing integrity constraints over the global schema, is an important novel feature provided by our system, that is missed by all the commercial tools we have considered in this work.

Integrity constraints help the designers of a data integration system in suitably modelling their domain of interest. In this respect we provided a way to treat three of the most important kinds of integrity constraints used in databases: key dependencies, inclusion dependencies (and hence foreign keys) and exclusion dependencies. The study of consistent query answering for such constraints led us to the definition of classes of queries for which the problem of computing the consistent answers in tractable, and in particular, polynomial in data complexity. More in detail, as a central feature, our system is able to perform consistent query answering directly over the (possibly inconsistent) data sources, by rewriting the user query together with the integrity constraints into first-order logic queries, then translated in SQL and directly evaluated against the data sources wrapped by the system.

All the techniques proposed have been tested on a real large database, and the results of such experiments show the effective feasibility of the whole system.

Of course, several research topics still remain open. In particular, among the others, we believe the following research lines are particularly important:

- extending the first-order techniques for consistent query answering presented in this work towards more powerful query languages and more expressive forms of integrity constraints;
- optimizing the first-order queries produced by the rewriting algorithms presented in Chapter 6 and, in turn, of the corresponding SQL queries, in order to improve the performance of the overall query answering technique (see e.g. [46] for important results in this directions);
- investigate the conditions under which one of the two techniques presented in Chapter 5 to deal with the mapping is preferable with respect to the other, trying to detect at query time whether exploit the underlying DBMS or the unfolding-based technique.

Bibliography

[1] The TPC-H desicion support benchmark. `http://www.tpc.org/tpch/` (Link checked on october 2006).

[2] IBM websphere information integrator: Accessing and integrating diverse data for the on demand business. *IBM White Paper* (2005).

[3] Information infrastructure: delivering the advantage of information integration today. *IBM White Paper* (2005).

[4] Abiteboul, S., and Duschka, O. Complexity of answering queries using materialized views. In *Proceedings of the Seventeenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'98)* (1998), pp. 254–265.

[5] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison Wesley Publ. Co., Reading, Massachussetts, 1995.

[6] Andritsos, P., Fuxman, A., and Miller, R. J. Clean answers over dirty databases: A probabilistic approach. In *International Conference on Data Engeneering* (2006).

[7] Arenas, M., Bertossi, L. E., and Chomicki, J. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'99)* (1999), pp. 68–79.

[8] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., Eds. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[9] Beneventano, D., Bergamaschi, S., Castano, S., Corni, A., Guidetti, R., Malvezzi, G., Melchiori, M., and Vincini, M. Information integration: the MOMIS project demonstration. In *Proceedings of the Twentysixth International Conference on Very Large Data Bases (VLDB 2000)* (2000).

[10] Bergamaschi, S., Castano, S., Vincini, M., and Beneventano, D. Semantic integration of heterogeneous information sources. *Data and Knowledge Engineering 36*, 3 (2001), 215–249.

[11] Bernstein, P. A., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. Data management for peer-to-peer computing: A vision. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)* (2002).

[12] Bouzeghoub, M., Fabret, F., Galhardas, H., Matulovic-Broqué, M., Pereira, J., and Simon, E. Data warehouse refreshment. In *Fundamentals of Data Warehouses*, M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis, Eds. Springer, 1999, pp. 47–86.

[13] Bouzeghoub, M., and Lenzerini, M. Introduction to the special issue on data extraction, cleaning, and reconciliation. *Information Systems 26*, 8 (2001), 535–536.

[14] Bravo, L., and Bertossi, L. Logic programming for consistently querying data integration systems. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)* (2003), pp. 10–15.

[15] Bruni, P., Arnaudies, F., Bennett, A., Englert, S., and Keplinger, G. Data federation with IBM DB2 Information Integrator V8.1, 2003. Redbook, `http://www.redbooks.ibm.com/redbooks/pdfs/sg247052.pdf`.

[16] Calì, A., Calvanese, D., De Giacomo, G., and Lenzerini, M. Data integration under integrity constraints. In *Proceedings of the Fourteenth International Conference on Advanced Information Systems Engineering (CAiSE 2002)* (2002), vol. 2348 of *Lecture Notes in Computer Science*, Springer, pp. 262–279.

[17] Calì, A., Calvanese, D., De Giacomo, G., Lenzerini, M., Naggar, P., and Vernacotola, F. IBIS: Semantic data integration at work. In *Proceedings of the Fifteenth International Conference on Advanced Information Systems Engineering (CAiSE 2003)* (2003), pp. 79–94.

[18] Calì, A., De Nigris, S., Lembo, D., Messineo, G., Rosati, R., and Ruzzi, M. DIS@DIS: a system for semantic data integration under integrity constraints. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)* (2003), pp. 335–338.

[19] Calì, A., Lembo, D., and Rosati, R. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proceedings of the Twentysecond ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2003)* (2003), pp. 260–271.

[20] Calì, A., Lembo, D., and Rosati, R. Query rewriting and answering under constraints in data integration systems. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)* (2003), pp. 16–21.

[21] Calì, A., Lembo, D., and Rosati, R. A comprehensive semantic framework for data integration systems. *Journal of Applied Logic 3*, 2 (2005), 308–328.

[22] Calì, A., Lembo, D., Rosati, R., and Ruzzi, M. Experimenting data integration with DIS@DIS. In *Proceedings of the Sixteenth International Conference on Advanced Information Systems Engineering (CAiSE 2004)* (2004), vol. 3084 of *Lecture Notes in Computer Science*, Springer, pp. 51–56.

[23] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. What to ask to a peer: Ontology-based query reformulation. In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR 2004)* (2004), pp. 469–478.

[24] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. Inconsistency tolerance in P2P data integration: an epistemic logic approach. In *Proc. of the Tenth International Symposium on Database Programming Languages (DBPL 2005)* (2005), pp. 90–105.

[25] Calvanese, D., De Giacomo, G., and Lenzerini, M. Answering queries using views over description logics knowledge bases. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)* (2000), pp. 386–391.

[26] Calvanese, D., De Giacomo, G., Lenzerini, M., and Rosati, R. Logical foundations of peer-to-peer data integration. In *Proceedings of the Twentythird ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2004)* (2004), pp. 241–251.

[27] Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R., and Vetere, G. DL-Lite: Practical reasoning for rich DLs. In *Proceedings of the 2004 Description Logic Workshop (DL 2004)* (2004), CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-104/`.

[28] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. Y. Answering regular path queries using views. In *Proceedings of the Sixteenth IEEE International Conference on Data Engineering (ICDE 2000)* (2000), pp. 389–398.

[29] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. Y. Query processing using views for regular path queries with inverse. In *Proceedings of the Nineteenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2000)* (2000), pp. 58–66.

[30] Carey, M. J., Haas, L. M., Schwarz, P. M., Arya, M., Cody, W. F., Fagin, R., Flickner, M., Luniewski, A., Niblack, W., Petkovic, D., Thomas, J., Williams, J. H., and Wimmers, E. L. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. of the 5th Int. Workshop on Research Issues in Data Engineering – Distributed Object Management (RIDE-DOM'95)* (1995), IEEE Computer Society Press, pp. 124–131.

[31] Cattell, R. G. G., and Barry, D. K., Eds. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, Los Altos, 1997.

[32] Chaudhuri, S., Krishnamurthy, S., Potarnianos, S., and Shim, K. Optimizing queries with materialized views. In *Proceedings of the Eleventh IEEE International Conference on Data Engineering (ICDE'95)* (1995), pp. 190–200.

[33] Chawathe, S. S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J. D., and Widom, J. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. of the 10th Meeting of the Information Processing Society of Japan (IPSJ'94)* (1994), pp. 7–18.

[34] Chomicki, J., and Marcinkowski, J. Minimal-change integrity maintenance using tuple deletions. Tech. Rep. cs.DB/0212004 v1, arXiv.org e-Print archive, Dec. 2002. Available at `http://arxiv.org/abs/cs/0212004`.

[35] Chomicki, J., and Marcinkowski, J. On the computational complexity of minimal-change integrity maintenance in relational databases. In *Inconsistency Tolerance*, L. Bertossi, A. Hunter, and T. Schaub, Eds., vol. 3300 of *Lecture Notes in Computer Science*. Springer, 2005, pp. 119–150.

[36] Chomicki, J., Marcinkowski, J., and Staworko, S. Computing consistent query answers using conflict hypergraphs. In *Proceedings of the Thirteenth International Conference on Information and Knowledge Management (CIKM 2004)* (2004), pp. 417–426.

[37] Chomicki, J., Marcinkowski, J., and Staworko, S. Hippo: a system for computing consistent query answers to a class of SQL queries. In *Proceedings of the Ninth International Conference on Extending Database Technology (EDBT 2004)* (2004), Springer, pp. 841–844.

[38] Dalvi, N., and Suciu, D. Efficient query evaluation on probabilistic databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004)* (2004).

[39] Duschka, O. M., and Genesereth, M. R. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'97)* (1997), pp. 109–116.

[40] Duschka, O. M., Genesereth, M. R., and Levy, A. Y. Recursive query plans for data integration. *Journal of Logic Programming 43*, 1 (2000), 49–73.

[41] Eiter, T., Fink, M., Greco, G., and Lembo, D. Efficient evaluation of logic programs for querying data integration systems. pp. 163–177.

[42] Eiter, T., Gottlob, G., and Mannilla, H. Disjunctive Datalog. *ACM Transactions on Database Systems 22*, 3 (1997), 364–418.

[43] Fagin, R., Kolaitis, P. G., Miller, R. J., and Popa, L. Data exchange: Semantics and query answering. In *Proceedings of the Ninth International Conference on Database Theory (ICDT 2003)* (2003), pp. 207–224.

[44] Fagin, R., Kolaitis, P. G., and Popa, L. Data exchange: Getting to the core. In *Proceedings of the Twentysecond ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2003)* (2003), pp. 90–101.

[45] Friedman, M., Levy, A., and Millstein, T. Navigational plans for data integration. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)* (1999), AAAI Press/The MIT Press, pp. 67–73.

[46] Fuxman, A., Fazli, E., and Miller, R. J. ConQuer: Efficient management of inconsistent databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2005), pp. 155–166.

[47] Fuxman, A., and Miller, R. J. First-order query rewriting for inconsistent databases. In *Proceedings of the Tenth International Conference on Database Theory (ICDT 2005)* (2005), vol. 3363 of *LNCS*, Springer, pp. 337–351.

[48] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, Ca, 1979.

[49] Genereseth, M. R., Keller, A. M., and Duschka, O. M. Infomaster: An information integration system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1997), pp. 539–542.

[50] Grahne, G., and Mendelzon, A. O. Tableau techniques for querying information sources through global schemas. In *Proceedings of the Seventh International Conference on Database Theory (ICDT'99)* (1999), vol. 1540 of *Lecture Notes in Computer Science*, Springer, pp. 332–347.

[51] Greco, G., Greco, S., and Zumpano, E. A logical framework for querying and repairing inconsistent databases. *IEEE Transactions on Knowledge and Data Engineering 15*, 6 (2003), 1389–1408.

[52] Gribble, S., Halevy, A., Ives, Z., Rodrig, M., and Suciu, D. What can databases do for peer-to-peer? In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB 2001)* (2001).

[53] Gryz, J. An algorithm for query folding with functional dependencies. In *Proc. of the 7th Int. Symp. on Intelligent Information Systems* (1998), pp. 7–16.

[54] Gryz, J. Query folding with inclusion dependencies. In *Proceedings of the Fourteenth IEEE International Conference on Data Engineering (ICDE'98)* (1998), pp. 126–133.

[55] Gupta, H., Harinarayan, V., Rajaraman, A., and Ullman, J. D. Index selection for OLAP. In *Proceedings of the Thirteenth IEEE International Conference on Data Engineering (ICDE'97)* (1997), pp. 208–219.

[56] Haas, L. A researcher's dream. *DB2 Magazine 8*, 3 (2003), 34–40.

[57] Haas, L., Lin, E., and Roth, M. Data integration through database federation. *IBM Systems Journal 41*, 4 (2002), 578–596.

[58] Halevy, A., Ives, Z., Suciu, D., and Tatarinov, I. Schema mediation in peer data management systems. In *Proceedings of the Nineteenth IEEE International Conference on Data Engineering (ICDE 2003)* (2003), pp. 505–516.

[59] Halevy, A. Y. Theory of answering queries using views. *SIGMOD Record 29*, 4 (2000), 40–47.

[60] Halevy, A. Y. Answering queries using views: A survey. *Very Large Database Journal 10*, 4 (2001), 270–294.

[61] Harinarayan, V., Rajaraman, A., and Ullman, J. D. Implemeting data cubes efficiently. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1996), pp. 205–216.

[62] Hull, R. Managing semantic heterogeneity in databases: A theoretical perspective. In *Proceedings of the Sixteenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'97)* (1997), pp. 51–61.

[63] Hull, R., and Zhou, G. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1996), pp. 481–492.

[64] IBM. `http://www.ibm.com/db2`. (link checked on october 2006).

[65] Keller, A. M., and Basu, J. A predicate-based caching scheme for client-server database architectures. *Very Large Database Journal 5*, 1 (1996), 35–47.

[66] Kirk, T., Levy, A. Y., Sagiv, Y., and Srivastava, D. The Information Manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments* (1995), pp. 85–91.

[67] Kwok, C. T., and Weld, D. Planning to gather information. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)* (1996), pp. 32–39.

[68] Lembo, D. Dealing with inconsistency and incompleteness in data integration, 2003.

[69] Lembo, L. G. D., Rosati, R., and Ruzzi, M. Consistent query answering under key and exclusion dependencies: algorithms and experiments. In *ACM Fourteenth Conference on Knowledge and Information Management, CIKM* (2005).

[70] Lenzerini, M. Data integration: A theoretical perspective. In *Proceedings of the Twentyfirst ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2002)* (2002), pp. 233–246.

[71] Leone, N., Greco, G., Ianni, G., Lio, V., Terracina, G., Eiter, T., Faber, W., Fink, M., Gottlob, G., Rosati, R., Lembo, D., Lenzerini, M., Ruzzi, M., Kalka, E., Nowicki, B., and Stani, W. The infomix system for advanced integration of incomplete and inconsistent data. In *SIGMOD Conference* (2005).

[72] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* (2005).

[73] Levy, A., Fikes, R. E., and Sagiv, S. Speeding up inference using relevance reasoning: A formalism and algorithms. *Artificial Intelligence 97*, 1–2 (1997).

[74] Levy, A. Y., Rajaraman, A., and Ordille, J. J. Query answering algorithms for information agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)* (1996), pp. 40–47.

[75] Levy, A. Y., Rajaraman, A., and Ordille, J. J. Querying heterogenous information sources using source descriptions. In *Proceedings of the Twentysecond International Conference on Very Large Data Bases (VLDB'96)* (1996).

[76] Levy, A. Y., Srivastava, D., and Kirk, T. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems 5* (1995), 121–143.

[77] Lloyd, J. W. *Foundations of Logic Programming (Second, Extended Edition)*. Springer, Berlin, Heidelberg, 1987.

[78] Manolescu, I., Florescu, D., and Kossmann, D. Answering XML queries on heterogeneous data sources. In *Proceedings of the Twentyseventh International Conference on Very Large Data Bases (VLDB 2001)* (2001), pp. 241–250.

[79] Microsoft. `http://www.microsoft.com/sqlserver`. (link checked on october 2006).

[80] Mitra, P. An algorithm for answering queries efficiently using views. Tech. rep., University of Southern California, Information Science Institute, Stanford (CA, USA), 1999. Available at `http://dbpubs.stanford.edu/pub/1999-46`.

[81] Niemelä, I., Simons, P., and Syrjänen, T. Smodels: A system for answer set programming. In *In Proceeding of Non Monotonic Reasoning 2000* (2000).

[82] Oracle. `http://www.oracle.con/database`. (link checked on october 2006).

[83] Papadimitriou, C. H. *Computational Complexity*. Addison Wesley Publ. Co., Reading, Massachussetts, 1994.

[84] Papakonstantinou, Y., Garcia-Molina, H., and Ullman, J. D. MedMaker: A mediation system based on declarative specifications. In *Proceedings of the Twelfth IEEE International Conference on Data Engineering (ICDE'96)* (1996), S. Y. W. Su, Ed., pp. 132–141.

[85] Poggi, A., and Ruzzi, M. Filling the gap between data integration and data federation. In *12th Italian Symposium on Advanced Database Systems, SEBD* (2004).

[86] Pottinger, R., and Levy, A. Y. A scalable algorithm for answering queries using views. In *Proceedings of the Twentysixth International Conference on Very Large Data Bases (VLDB 2000)* (2000), pp. 484–495.

[87] Rahm, E., and Do, H. H. Data cleaning: Problems and current approaches. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 23*, 4 (2000), 3–13.

[88] Savitch, W. J. Relationship between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences 4* (1970), 177–192.

[89] Serafini, L., Giunchiglia, F., Mylopoulos, J., and Bernstein, P. The local relational model: Model and proof theory, 2001.

[90] Theodoratos, D., and Sellis, T. Data warehouse design. In *Proceedings of the Twentythird International Conference on Very Large Data Bases (VLDB'97)* (1997), pp. 126–135.

[91] Tsatalos, O. G., Solomon, M. H., and Ioannidis, Y. E. The GMAP: A versatile tool for phyisical data independence. *Very Large Database Journal 5*, 2 (1996), 101–118.

[92] Ullman, J. D. Information integration using logical views. In *Proceedings of the Sixth International Conference on Database Theory (ICDT'97)* (1997), vol. 1186 of *Lecture Notes in Computer Science*, Springer, pp. 19–40.

[93] Ullman, J. D. Information integration using logical views. *Theoretical Computer Science 239*, 2 (2000), 189–210.

[94] Vardi, M. Y. The complexity of relational query languages. In *Proceedings of the Fourteenth ACM SIGACT Symposium on Theory of Computing (STOC'82)* (1982), pp. 137–146.

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

[96] Wiener, J. L., Gupta, H., Labio, W. J., Zhuge, Y., Garcia-Molina, H., and Widom, J. A system prototype for warehouse view maintenance. Tech. rep., Stanford University, 1996. Available at `http://www-db-stanford.edu/warehousing/warehouse.html`.

[97] Wijsen, J. Project-join-repair: An approach to consistent query answering under functional dependencies. In *Lecture Notes in Computer Science* (2006), vol. 4027, pp. 1–12.

[98] Yang, J., Karlapalem, K., and Li, Q. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the Twentythird International Conference on Very Large Data Bases (VLDB'97)* (1997), pp. 136–145.

[99] Yerneni, R., Li, C., Garcia-Molina, H., and Ullman, J. D. Computing capabilities of mediators. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1999), pp. 443–454.

[100] Zhou, G., Hull, R., and King, R. Generating data integration mediators that use materializations. *Journal of Intelligent Information Systems 6* (1996), 199–221.

[101] Zhou, G., Hull, R., King, R., and Franchitti, J.-C. Data integration and warehousing using H20. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 18*, 2 (1995), 29–40.

[102] Zhou, G., Hull, R., King, R., and Franchitti, J.-C. Using object matching and materialization to integrate heterogeneous databases. In *Proceedings of the Third International Conference on Cooperative Information Systems (CoopIS'95)* (1995), pp. 4–18.