OPTIMIZING AND IMPLEMENTING REPAIR PROGRAMS FOR CONSISTENT QUERY ANSWERING IN DATABASES

by

Mónica Caniupán

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

Ottawa-Carleton Institute for Computer Science

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario
March, 2007

Abstract

Databases may not always satisfy their integrity constraints (ICs) and a number of different reasons can be held accountable for this. However, in most cases an important part of the data is still consistent with the ICs, and can still be retrieved through queries posed to the database. Consistent query answers are characterized as ordinary answers obtained from every minimally repaired and consistent version of the database. Database repairs wrt a wide class of ICs can be specified as stable models of disjunctive logic programs. Thus, Consistent Query Answering (CQA) for first-order queries is translated into *cautious* reasoning under the stable models semantics.

The use of logic programs does not exceed the intrinsic complexity of CQA. However, using them in a straightforward manner is usually inefficient. The goal of this thesis is to develop optimized techniques to evaluate queries over inconsistent databases by using logic programs. More specifically, we optimize the structure of programs, model computation, and evaluation of queries from them. We develop a system which implements optimized logic programs and efficient methods to compute consistent answers to first-order queries.

Moreover, we propose the use of the well-founded semantics (WFS) as an alternative way to obtain consistent answers. We show that for a certain class of queries and ICs, the well founded interpretation of a program retrieves the same consistent answers as the stable models semantics. The WFS has lower data complexity than the stable models semantics.

We also extend the use of logic programs for retrieving consistent answers to aggregate queries, and we develop a repair semantics for Multidimensional Databases.

To my parents.

Acknowledgements

First, I would like to thank Leopoldo Bertossi, my supervisor. He is one of those persons that you never forget, he has been a big influence on my life. He has taught me not just logic, but how to be a better professional. Thanks for the "holistic" scholarship.

I would like to thank my family, specially my parents Domingo and Any, my sister Anita, and brother Enrique, for their unconditional love, support, and friendship. Also I would like to thank Fernando Torres, who was an important support at the beginning of my studies.

From the Universidad del Bío-Bío, I would like to thank Luis Contreras, and Patricio Galvez, for their support and motivation through all my studies. Thanks also to Oscar Gericke, and Alex Medina for their continuing support. Special thanks to Claudio Gutiérrez, Pedro Campos, Germán Poó, the secretaries María Rivera, and Marta Hermosilla.

From the Pontificia Universidad Catolica (PUC) of Chile, I would like to thank Alvaro Campos who was a very generous person, and unfortunately left us unexpectedly; to my friends of the logic group: Alvaro Cortés, Pablo Barcelo and Loreto Bravo. I'd like to specially thank Loreto, my dear friend for many years; we have spent too much time together, working, talking and laughing. Thanks to the administrative staff: María Soledad Carrión, Cecilia Venegas, and Alda Briceño.

From the School of Computer Science at Carleton University I would like to thank Doug Howe, the chair of the department, who is also a member of my thesis committee, Sivarama Dandamudi, who was part of my comprehensive, and PhD proposal committees and unfortunately died last year, Jean-Pierre Corriveau the Graduate

Director, Michiel Smid the chair for all my examination committees, Mengchi Liu, who was a member of my comprehensive, and PhD proposal committees. The administrative staff: Claire Ryan, Linda Pfeiffer, and Sharmila Namasivayampillai.

Thanks also to people from other universities with whom I had useful and enlightening conversations: Carlos Hurtado, Alejandro Vaisman, Alberto Mendelzon, Wolfgang Faber, Nicola Leone, and Amy Felty, who was in my comprehensive and PhD proposal committees. To the members of my final dissertation committee: Iluju Kiringa, Jarek Gryz, and Alex Ramirez.

Finally, I'd like to thank my dear friends who shared with me during all the years I spent in Ottawa: Martin Jones, Jennifer Gisseleire, Paloma Bertossi, Carla Corral, Leo Ferres, Gerardo Reynaga, Miguel Vargas, Natalia Villanueva, Mauricio Vines, Carolina Ulloa, Emilie Paquin-Holmested, José Miguel Cruz, Joaquín Aristimuño.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Integrity constraints (ICs) play an important role in databases. They capture the intended meaning (semantics) of the data in the database. Nevertheless, databases may become inconsistent wrt ICs due to several reasons:

(a) In virtual data integration [54] of multiple data sources, that are possibly individually consistent regarding local ICs, the system may become inconsistent wrt the global ICs.

(b) A stand alone relational database management system may not have mechanisms to express/maintain certain ICs.

(c) In legacy systems, data may not satisfy new semantic constraints.

(d) In the presence of user or informational constraints, which are used, but not necessarily enforced by the system, etc.

Even though, ICs may be violated by databases, in most cases only a small portion of the data is inconsistent wrt those ICs. Moreover, an inconsistent database can still give us useful information. In addition, it could be impossible or simply undesirable to restore consistency of the database, because (a) There is no permission to modify the data. (b) It is an expensive process. (c) There are no mechanisms to enforce the ICs. (d) It can be better for performance purposes not to enforce them, etc.

Thus, if the inconsistent database is going to be queried we need to distinguish between the data (database tuples) that is affected by ICs violations and the data that is not. Therefore, it becomes necessary and reasonable to develop methods for retrieving consistent answers to queries from inconsistent databases.

The notion of consistent answer to a first-order (FO) query was initially defined in [2], along with a mechanism for computing them. Intuitively, a ground tuple $\bar{t}$ is a consistent answer to a query $Q(\bar{x})$ in a database instance D if it is an (ordinary) answer to $Q(\bar{x})$ in every minimal, under set inclusion, repair of D. A repair is a database instance D' over the same schema that is obtained from D by deleting or inserting whole database tuples, satisfies the ICs, and departs minimally from D under set inclusion.

Example 1.1 Consider the database schema *Student*$(Name, Depart)$. The functional dependency (FD) $Name \rightarrow Depart$ establishes that each student name is associated with a unique department value. The first two tuples of the following database instance violate the functional dependency:

Student	Name	Depart
	smith	*cs*
	smith	*math*
	jones	*math*

Consistency can be minimally restored by deleting either tuple *Student*(*smith*, *cs*) or tuple *Student*(*smith*, *math*). If we delete both tuples, the resulting database is not a repair since it does not satisfy minimality. Therefore, there are two database repairs:

Student	Name	Depart
	smith	*cs*
	jones	*math*

Student	Name	Depart
	smith	*math*
	jones	*math*

We can see that certain information persists in the repairs, e.g. tuple *Student*(*jones*, *math*) is in both repairs, since it does not violate the FD. On the other hand, the inconsistent tuples *Student*(*smith*, *cs*) and *Student*(*smith*, *math*) do not persist in all the repairs. If we want to know the name of the students in the *math* department, we can pose the query *Student*(x, *math*). The answer to this query is (*jones*) in both repairs, therefore the consistent answer is (*jones*).

Moreover, for the boolean disjunctive query *Student*(*smith*, *cs*) $\vee$ *Student* (*smith*, *math*) the consistent answer is *yes*, since both database repairs satisfy either one of the tuples in the query. Notice that if we had deleted all the inconsistent data, we would have lost this information. □

Even though the notion of consistent answer is given in terms of database repairs, it does not mean that we need to compute all the database repairs to obtain consistent answers. Actually, the computation of all the database repairs could be exponential, as it is illustrated in the example below.

Example 1.2 [10] Consider the database schema $R(A, B)$ with FD $A \rightarrow B$, and the following database instance that is inconsistent wrt the FD:

R	A	B
	1	0
	1	1
	2	0
	2	1
	.	.
	n	0
	n	1

There are 2^n database repairs, hence it is not viable to evaluate queries by computing all the database repairs. □

Since the computation of repairs seems to be non practical in some cases, different methods to compute consistent answers that avoid the explicit computation of database repairs have been proposed in the literature. The methods can be classified into two groups [10]: methods based on *query transformation*, and methods that use a *compact representation of repairs*.

1) *Query Transformation*: Given a FO query $\mathcal{Q}$ and a set of ICs, generate a new FO query $\mathcal{Q}$' such that when posed to a database, its usual answers correspond to the consistent answers to $\mathcal{Q}$ wrt the ICs.

The first method for computing consistent answers to first-order queries based on query transformation was proposed in [2], and implemented in [22]. This method works for quantifier free conjunctive queries, and for a restricted set of ICs, such as functional dependencies, and full set inclusion dependencies. However, it does not consider more general queries or ICs with existential quantifiers, like referential ICs. This method is polynomial in data complexity. Other polynomial time methods based on query rewriting are presented in [27, 39], which also work for restricted set of ICs and classes of conjunctive queries with limited forms of projection.

2) *Compact Representation of Repairs:* Given a database instance and a set of ICs, the goal is to construct an efficient representation of all the repairs for the database instance, and use it to answer queries.

Compact representations of repairs were presented first in [4, 5], where repairs wrt functional dependencies are represented as maximal independent sets in conflict graphs. This approach was also adopted in [26, 27] to represent database repairs regarding denial constraints, and compute consistent answers to quantifier free first-order queries.

Another form of compact representation of repairs, which moreover permits to handle Consistent Query Answering (CQA) for more complex classes of queries and

ICs, is the representation of database repairs as stable models [44, 68] of disjunctive logic programs [33]. The latter act as an executable and compact logical specifications of repairs. Disjunctive logic programs [33] have been introduced as a tool for knowledge representation and non-monotonic reasoning. The stable models semantics [44, 68] is the most accepted semantics for this kind of programs.

Disjunctive repair programs with stable models semantics [44, 68] were first introduced in [3, 46] to specify database repairs. Simpler and more general repair programs were later introduced in [6, 7] for CQA. The logic programming approach is more general than the methods presented before. It works for all universal ICs and queries that are first-order or even expressed in Datalog with negation. In [12] the methodology was extended to handle sets of acyclic referential ICs. Later, in [16] the methodology was extended to handle databases that may contain null values. In [7, 16] it was shown that there is a one to one correspondence between the stable models of the repair program and the database repairs wrt *RIC*-acyclic sets of ICs [16] (cf. Definition 2.2).

Moreover, logic-based approaches to CQA in the context of data integration systems were presented in [11, 15, 34, 55]. In a data integration system, a set of independent databases, possibly individually consistent wrt local ICs, are integrated into a new, global, virtual database. This new global database may become inconsistent wrt different global ICs. In [13] a logic programming framework for CQA in peer-to-peer data exchange systems is described.

In the general case, the data complexity of CQA is Π^P_2-complete [28]. Consequently, in the worst case, it is necessary to use an expressive language such as disjunctive logic programs under stable models semantics, for which query evaluation has the same data complexity [30]. Nevertheless, it is possible to identify classes of ICs and queries, for which CQA has lower data complexity. Some polynomial time cases of CQA have been identified in [2, 7, 27, 39].

In this thesis, we are interested in optimizing and implementing the general logic programming approach to CQA from stand alone databases. Essentially, we are concerned with improving the structure of repair programs (cf. Chapter 4), model computation, and query evaluation from them (cf. Chapter 5). For instance, we develop optimization techniques that reduce the amount of data involved in the computation of queries, in such a way that only relevant base data and program rules are involved in query evaluation.

The optimized techniques presented in Chapter 5 are implemented in the "Consistency Extractor System", which is described in Chapter 9. Also, in Chapter 9 we report experimental results that show the gain, in terms of execution time of queries, of our optimized methods for computing query answers.

We also explore the use of the well-founded semantics of programs [75] to CQA. This semantics has lower data complexity than the stable models semantics, and therefore, it becomes a good alternative for cases in which CQA has lower data complexity. The analysis of the WFS of programs is done in Chapter 7.

Moreover, we extend logic programs to compute consistent answers to aggregate queries with *scalar* functions, and aggregate queries with *group-by* statements. This is presented in Chapter 6.

Scalar queries apply one of the aggregate functions *min, max, count, sum, avg* over an attribute of a database relation, and return a unique value for the whole relation. The group-by queries perform grouping on the values of an attribute or a set of attributes, and apply one of the aggregate functions over each group. In this way, a single value for each group is computed, instead of a unique value for the whole relation. The semantics of consistent answers to *scalar* aggregate queries was given in [4]. The semantics of consistent answers to aggregate queries with *group-by* statements adopted in this thesis follows the spirit of the semantics presented in [4].

Moreover, we develop a repair semantics for Multidimensional Databases (MDBs) [48]. This kind of databases differs from the relational databases considered so far in terms of data schema and ICs. We show that multidimensional ICs can easily be violated and affect the processing of queries. As a consequence, a new framework to deal with inconsistencies and retrieve consistent answers is needed for Multidimensional Databases.

This thesis is organized as follows: In Chapter 2 we recall some concepts and terminology. Chapter 3 presents the statement of the problem and the thesis contributions. Chapter 4 contains the optimizations performed on the structure of repair programs. Chapter 5 presents optimized methods to evaluate programs. In Chapter 6 we describe the specification of repair programs to compute consistent answers to aggregate queries. Chapter 7 studies the well-founded semantics of programs for CQA. Chapter 8 presents the repair semantics for Multidimensional Databases. Chapter 9 describes the "Consistency Extractor System", which is an optimized implementation of CQA based on logic programs. Also, this chapter presents experimental results. Final conclusions, future work, open problems, and contributions are discussed in Chapter 10.

Chapter 2

Background

In this chapter we recall the main concepts and terminology related to disjunctive Datalog programs and the stable models semantics. We also introduce basic concepts of databases, integrity constraints, database repairs, and consistent query answering.

2.1 Disjunctive Datalog Programs and Stable Model Semantics

A disjunctive Datalog rule r is an expression of the form:

$$a_1(\bar{x}_1) \vee \cdots \vee a_n(\bar{x}_n) \leftarrow b_1(\bar{y}_1), \ldots, b_k(\bar{y}_k),\ \mathit{not}\ b_{k+1}(\bar{y}_{k+1}), \ldots,\ \mathit{not}\ b_m(\bar{y}_m), \quad (2.1)$$

where $a_i(\bar{x}_i), b_i(\bar{y}_i)$ are *atoms* in a relational first-order language, a_i, b_i are relation names (predicates), and $\bar{x}_i, \bar{y}_i$ are lists of variables and constants that match the arity, i.e. number of attributes, of a_i, b_i respectively. Here, *not* represents weak negation, also known as *default* negation, and the symbol "," is conjunction. A *literal* is an atom, e.g. $p(\bar{c})$ or its negation $\mathit{not}\ p(\bar{c})$. For a set of literals $\mathcal{L}$, $\mathit{not}.\mathcal{L}$ is the set of literals that are complementary to those in $\mathcal{L}$. For a literal L, $\mathit{not}.L$ denotes the literal that is complementary to L.

The disjunction $a_1(\bar{x}_1) \vee \cdots \vee a_n(\bar{x}_n)$ is the *head* of r, and it is denoted by $H(r)$, while the conjunction $b_1(\bar{y}_1), \ldots, b_k(\bar{y}_k),\ \mathit{not}\ b_{k+1}(\bar{y}_{k+1}), \ldots,\ \mathit{not}\ b_m(\bar{y}_m)$ is the *body* of r, denoted by $B(r)$. We identify with $B^+(r)$ the set of positive literals occurring in the body of the rule i.e. $B^+(r) = \{b_1(\bar{y}_1), \ldots, b_k(\bar{y}_k)\}$, and $B^-(r)$ denotes the set of complements of negative literals in the body of r i.e. $B^-(r) = \{b_{k+1}(\bar{y}_{k+1}), \ldots, b_m(\bar{y}_m)\}$.

A disjunctive program is a finite set of rules. If the rules do not involve disjunction, i.e. $n = 1$ for every rule in the program, then the program is called a *disjunction free* or *normal* program. If $k = m = 0$, then r is a fact, and we omit the symbol $\leftarrow$. A program is called *positive* if $m = 0$ for every rule. For a disjunctive Datalog program Π and set of facts D, we denote by $\Pi[D]$ the program $\Pi \cup D$.

We call *extensional* predicates to the predicates (EDB) that occur only in the body of the rules i.e. they are defined by the facts of a database. Otherwise they are called *intensional* predicates (IDB), i.e. predicates that are defined by the rules in the program.

For any program Π, the *Herbrand* universe [64], denoted by $\mathcal{U}_\Pi$, is the set of all constants appearing in Π. In case no constant appears in the program, an arbitrary constant is added to $\mathcal{U}_\Pi$. The *Herbrand* base, $\mathcal{B}_\Pi$, is the set of all ground atoms constructible from the predicate symbols appearing in Π and the constants of $\mathcal{U}_\Pi$. The Herbrand instantiation of a program Π corresponds to all the instances of rules of the form (2.1) in Π, where variables are replaced with constants from $\mathcal{U}_\Pi$. We denote by $ground(\Pi)$ the instantiated or *ground* program Π. A ground program is also called a *propositional* program.

An interpretation I for a program Π is a subset $I \subseteq \mathcal{B}_\Pi$. A ground atom $p(\bar{c})$ is true wrt I if $p(\bar{c}) \in I$, and false otherwise. A ground rule r of the form (2.1) is satisfied by I if either: (a) some of the atoms in $H(r)$ is in I i.e. $H(r) \cap I \neq \emptyset$, or (b) some of the negative literals in the body of r is in I, i.e. $B^-(r) \cap I \neq \emptyset$, or (c) some of the positive literals in the body of r i.e. some literal in $B^+(r)$, is false wrt I. Otherwise, the rule is not satisfied. The interpretation I is a *model* of program Π if it satisfies all the rules in $ground(\Pi)$. Moreover, a model for $ground(\Pi)$ is *minimal* if no proper subset of it is a model of *ground*($\mathit{\Pi}$), i.e. if no proper subset of it satisfies all the rules in *ground*($\mathit{\Pi}$).

We adopt the stable models semantics [44, 68] as the semantics for a disjunctive Datalog program. According to it, a disjunctive program can have several minimal stable models. An interpretation I is a *stable* model of program Π if it is a minimal model of the *Gelfond-Lifschitz* (GL) reduction of Π wrt I, such a reduction is an instantiated program without negation obtained from *ground*($\mathit{\Pi}$) by: (a) deleting all the rules having a ground literal of the form $\;not\ p(\bar{c})$ with $p(\bar{c}) \in I$, (b) eliminating all the ground literals of the form $\;not\ p(\bar{c})$ from the remaining rules. The set of all the stable models of a program Π is denoted by $SM(\Pi)$.

Example 2.1 For program Π: $M(x) \vee Q(x) \leftarrow R(x),\ not\ S(x)$, with $\mathcal{U} = \{a\}$, and interpretation $I = \{R(a), M(a)\}$, the ground program *ground*($\mathit{\Pi}$) is: $M(a) \vee Q(a) \leftarrow R(a),\ not\ S(a)$. The GL reduction of Π regarding I is: $M(a) \vee Q(a) \leftarrow R(a)$. Here interpretation I is a minimal model of program Π, and therefore $\{R(a), M(a)\}$ is a stable model of Π. Program Π has two stable models: $\mathcal{M}_1 = \{R(a), M(a)\}$ and $\mathcal{M}_2 = \{R(a), Q(a)\}$. □

In our setting, stable models are sets of ground atoms. They are total interpretations, i.e. atoms are either true or false in the model, but never unknown. More specifically, any ground atom that is not in the model is considered false, by applying the *closed world assumption* [69] to the model. For instance, in Example 2.1 atom $Q(a)$ is not in stable model $\mathcal{M}_1$, and therefore it is considered false regarding $\mathcal{M}_1$. Also, atom $M(a)$ is false wrt $\mathcal{M}_2$, and atom $S(a)$ is false regarding both stable models.

In the logic programming framework under stable models semantics, there are two main notions of reasoning, the *cautious* and the *brave* reasoning. In the former, a literal L is entailed by the program if it is true in every stable model of the program.

In the latter, a literal L is entailed by the program if it is true in at least one stable model of the program. For instance, in Example 2.1, atom $R(a)$ is cautiously true, since $R(a) \in \mathcal{M}_1$ and $R(a) \in \mathcal{M}_2$. On the contrary, atom $Q(a)$ is only bravely true since it is true only in model $\mathcal{M}_2$.

In addition, two programs Π_1 and Π_2 are bravely (resp. cautiously) equivalent wrt a query $\mathcal{Q}$, denoted $\Pi_1 \equiv_{\mathcal{Q}} \Pi_2$, if for any set D of facts, brave (resp. cautious) answers to $\mathcal{Q}$ from the program $\Pi_1 \cup D$ are the same as the brave (resp. cautious) answers to $\mathcal{Q}$ from $\Pi_2 \cup D$.

2.2 Databases and Integrity Constraints

We consider a relational database schema $\Sigma = (\mathcal{U}, \mathcal{R}, \mathcal{B})$, where $\mathcal{U}$ is the possibly infinite database domain with *null* $\in \mathcal{U}$, $\mathcal{R}$ is a fixed set of database predicates, each of them with a finite, and ordered set of attributes, and $\mathcal{B}$ is a fixed set of built-in predicates, like comparison predicates, e.g. $\{<, >, =, \neq\}$. $R[i]$ denotes the attribute in position i of predicate $R \in \mathcal{R}$. Database instances of a relational schema Σ are finite collections D of ground atoms of the form $R(c_1, ..., c_n)$, which are called *database tuples*, where $R \in \mathcal{R}$, and $(c_1, ..., c_n)$ is a *tuple* of constants, i.e. elements of $\mathcal{U}$. The extensions for built-in predicates are fixed, and possibly infinite in every database instance. There is also a fixed set IC of integrity constraints, that are sentences in the first-order language $\mathcal{L}(\Sigma)$ determined by Σ. They are expected to be satisfied by any database instance of Σ, but they may not.

An *integrity constraint* is a sentence $\psi \in \mathcal{L}(\Sigma)$ of the form:

$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \;\longrightarrow\; \exists \bar{z}(\bigvee_{j=1}^{n} Q_j(\bar{y}_j, \bar{z}_j) \vee \varphi)), \tag{2.2}$$

where $P_i, Q_j \in \mathcal{R}$, $\bar{x} = \bigcup_{i=1}^{m} \bar{x}_i$, $\bar{z} = \bigcup_{j=1}^{n} \bar{z}_j$, $\bar{y}_j \subseteq \bar{x}$, $\bar{x} \cap \bar{z} = \emptyset$, $\bar{z}_i \cap \bar{z}_j = \emptyset$ for

$i \neq j$, and $m \geq 1$.[1] Here φ is a formula containing only disjunctions of built-in atoms from $\mathcal{B}$ whose variables appear in the antecedent of the implication.[2] We will assume that there exists a propositional atom **false** $\in \mathcal{B}$ that is always false in the database. Domain constants different from *null* may appear in a constraint of the form (2.2). In this thesis we will consider universal and referential ICs. When writing them, we usually will not write the prefix of universal quantifiers.

A *universal integrity constraint* (UIC) is a sentence in $\mathcal{L}(\Sigma)$ that is logically equivalent to a sentence of the form:

$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \ \rightarrow \ \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi), \tag{2.3}$$

that is, a formula of the form (2.2) with $\bar{z} = \emptyset$, i.e. without existentially quantified variables.

A *referential integrity constraint* (RIC) is a sentence of the form:[3]

$$\forall \bar{x}(P(\bar{x}) \rightarrow \exists \bar{z}\ Q(\bar{y}, \bar{z})), \tag{2.4}$$

that is, a formula of the form (2.2) with $m = n = 1$, $\varphi = \emptyset$, $\bar{y} \subseteq \bar{x}$, and $P, Q \in \mathcal{R}$.

The class of ICs of the form (2.2) includes most of the ICs commonly found in database practice, e.g. a *denial constraint* can be expressed as $\bar{\forall}\bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \longrightarrow \textbf{false})$. Functional dependencies can be expressed by several implications of the form (2.2), each of them with a single equality in the consequent. Partial inclusion dependencies are RICs, and full inclusion dependencies are UICs. We can also specify unary constraints, also called *check constraints*, that allow to express conditions on each row of a table, so they can be formulated with one predicate in the antecedent of (2.2) and

[1]Note that if $\bar{z}_i \cap \bar{z}_j \neq \emptyset$ the formula can be rewritten as an equivalent formula such that $\bar{z}_i \cap \bar{z}_j = \emptyset$.

[2]The left hand side of a implication is called the antecedent, while that the right hand side is called the *consequent* of the implication.

[3]For simplification purposes, we assume that the existential variables appear in the last attributes of Q, but they may appear anywhere else in Q.

only a formula φ in the consequent. For example, $\forall xy(P(x, y) \rightarrow y > 0)$ is a unary constraint.

Example 2.2 For $\Sigma = \{\mathit{Student}(\mathit{id}, \mathit{name}),\ \mathit{Grad}(\mathit{id}, \mathit{name}), \mathit{Assistant}(\mathit{id}, \mathit{course})\}$, the following are UICs:

- The functional dependency (FD) $\mathit{Student}: \mathit{id} \rightarrow \mathit{name}$, expressed in $\mathcal{L}(\Sigma)$ by $\forall \mathit{id}\,\mathit{name}_1\,\mathit{name}_2\ (\mathit{Student}(\mathit{id}, \mathit{name}_1) \wedge \mathit{Student}(\mathit{id}, \mathit{name}_2) \rightarrow \mathit{name}_1 = \mathit{name}_2)$

- The full inclusion dependency (IND) $\mathit{Grad}[\mathit{id}, \mathit{name}] \subseteq \mathit{Student}[\mathit{id}, \mathit{name}]$, expressed by $\forall \mathit{id}\ \mathit{name}\ (\mathit{Grad}(\mathit{id}, \mathit{name}) \rightarrow \mathit{Student}(\mathit{id}, \mathit{name}))$.

The non-full inclusion dependency $\mathit{Assistant}[\mathit{id}] \subseteq \mathit{Student}[\mathit{id}]$ can be expressed as a RIC: $\forall \mathit{id}\ \mathit{course}(\mathit{Assistant}(\mathit{id}, \mathit{course}) \rightarrow \exists \mathit{name}\ \mathit{Student}(\mathit{id}, \mathit{name}))$. Here $\bar{x} = (\mathit{id}, \mathit{course})$, $\bar{y} = (\mathit{id})$, and $\bar{z} = (\mathit{name})$. □

We consider a fixed finite set IC of ICs of the form (2.2). Notice that sets of constraints of this form are always logically consistent, in the classical sense, since empty databases always satisfy them.

Definition 2.1 [21] The directed *dependency graph* $\mathcal{G}(\mathit{IC})$ for a set IC of ICs of the form (2.2) is defined as follow: each database predicate P in D is a node, and there is an edge (P_i, P_j) from P_i to P_j iff there exists a constraint $ic \in \mathit{IC}$ such that P_i appears in the antecedent of ic and P_j appears in the consequent of ic. In addition, there is an edge (P_i, P_i) from P_i to P_i if P_i appears in the antecedent of an ic which has only built-in predicates in its consequent. A node is called a *sink* (*source*) if it has only incoming (outgoing) edges. □

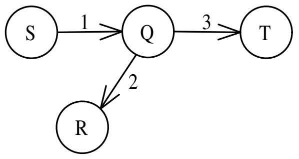


Figure 2.1: Dependency Graph $\mathcal{G}(IC)$

Example 2.3 Figure 2.1 shows the dependency graph $\mathcal{G}(IC)$ for the set IC of UICs containing $ic_1 = \forall x(S(x) \rightarrow Q(x))$ and $ic_2 = \forall x(Q(x) \rightarrow R(x))$, and the RIC $ic_3 = \forall x(Q(x) \rightarrow \exists y T(x,y))$.

Edges 1 and 2 correspond to the universal constraints ic_1 and ic_2 resp., and edge 3 to the referential IC. Nodes R and T are sink nodes, S is a source node. □

A *connected component* in a graph is a maximal subgraph such that, for every pair (A, B) of its vertices, there is a path from A to B or from B to A. For a graph $\mathcal{G}$, $\mathcal{C}(\mathcal{G}) := \{c \mid c \text{ is a connected component in } \mathcal{G}\}$; and $\mathcal{V}(\mathcal{G})$ is the set of vertices of $\mathcal{G}$.

Definition 2.2 [16] Given a set IC of UICs and RICs, IC_U denotes the set of UICs in IC. The *contracted dependency graph* of IC, $\mathcal{G}^C(IC)$, is obtained from $\mathcal{G}(IC)$ by replacing, for every $c \in \mathcal{C}(\mathcal{G}(IC_U))$,[4] the vertices in $\mathcal{V}(c)$ by a single vertex and deleting all the edges associated to the elements of IC_U. Finally, IC is said to be *RIC-acyclic* if $\mathcal{G}^C(IC)$ has no cycles. □

Example 2.4 (example 2.3 cont.) Figure 2.2 shows the contracted dependency graph $\mathcal{G}^C(IC)$, which is obtained by replacing in $\mathcal{G}(IC)$ the edges 1 and 2, and their end vertices by a vertex labelled with $\{Q, R, S\}$.

Since there are no loops in $\mathcal{G}^C(IC)$, the set IC is RIC-acyclic. However, if we add a new UIC: $\forall xy(T(x,y) \rightarrow R(y))$ to IC, all the vertices belong to the same connected component. Figure 2.3 shows $\mathcal{G}(IC)$ and $\mathcal{G}^C(IC)$, respectively. Since there is a self-loop in $\mathcal{G}^C(IC)$, the set of ICs is *not* RIC-acyclic.

[4]Notice that for every $c \in \mathcal{C}(\mathcal{G}(IC_U))$ it holds $c \in \mathcal{C}(\mathcal{G}(IC))$.

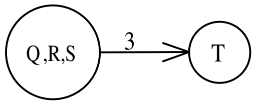


Figure 2.2: Contracted Dependence Graph $\mathcal{G}^C(IC)$

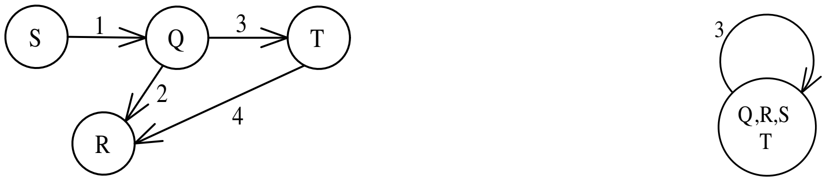


Figure 2.3: Non-RIC-acyclic Set of ICs

□

A set of UICs is always *RIC*-acyclic, as expected.

A database instance D is *consistent* if it satisfies the given set IC of ICs. Otherwise, it is *inconsistent* wrt IC. The semantics of constraint satisfaction in presence of null values we consider is the one defined in [16]. In order to present it, we need to introduce the concept of *relevant* attribute.

Definition 2.3 [16] For t a term, i.e. a variable or a domain constant, let $pos^R(\psi, t)$ be the set of positions in predicate $R \in \mathcal{R}$ where term t appears in IC ψ. The set $\mathcal{A}$ of *relevant attributes* for an IC ψ of the form (2.2) is:

$\mathcal{A}(\psi) = \{R[i] \mid x$ is variable present at least twice in ψ,[5] and $i \in pos^R(\psi, x)\} \cup$

$\{R[i] \mid c$ is a constant in ψ and $i \in pos^R(\psi, c)\}$,

where $R[i]$ denotes a position (or the correspondent attribute) in relation R. □

The relevant attributes include the attributes needed to check the satisfaction of the constraints, e.g. the attributes in joins, in built-ins, etc. Note that if the built-ins have variables that are redundant or not needed, it might have the undesirable effect

[5]If a variables appears at least twice in a IC, then it is involved in a join, or it is in the head and in the body of the IC, or it is in a built-in atom. In all these cases, the variable is relevant.

of transforming an attribute in relevant when it does not need to. For example the constraint $\forall xy(P(x,y) \rightarrow y > 3 \vee x = x)$ is equivalent to $\forall xy(P(x,y) \rightarrow y > 3)$, but the first has relevant attributes x and y and the second one, only y.

Definition 2.4 [16] For a set of attributes $\mathcal{A}$ and a predicate $P \in \mathcal{R}$, $P^{\mathcal{A}}$ denotes the predicate P restricted to the attributes in $\mathcal{A}$. $D^{\mathcal{A}}$ denotes the database D with all its database atoms projected onto the attributes in $\mathcal{A}$, i.e. $D^{\mathcal{A}} = \{P^{\mathcal{A}}(\Pi_{\mathcal{A}}(\bar{t})) \mid P(\bar{t}) \in D\}$, where $\Pi_{\mathcal{A}}(\bar{t})$ is the projection on $\mathcal{A}$ of tuple $\bar{t}$. $D^{\mathcal{A}}$ has the same underlying domain $\mathcal{U}$ as D. □

Example 2.5 Consider a UIC ψ : $\forall xyz(P(x,y,z) \rightarrow R(x,y))$ and the following database instance D:

P	X	Y	Z
	a	*b*	*null*
	b	*null*	*a*

R	X	Y
	a	*b*

Here x and y appear twice in ψ, therefore $\mathcal{A}(\psi) = \{P[1], R[1], P[2], R[2]\}$. The value in z is not relevant to check the satisfaction of the constraint, because we want to verify if the values in the first two attributes in P also appear in R, which is equivalent to checking if $\forall xy(P^{\mathcal{A}(\psi)}(x,y) \rightarrow R^{\mathcal{A}(\psi)}(x,y))$ is satisfied by $D^{\mathcal{A}(\psi)}$, where $D^{\mathcal{A}(\psi)}$ is:

$P^{\mathcal{A}(\psi)}$	X	Y
	a	*b*
	b	*null*

$R^{\mathcal{A}(\psi)}$	X	Y
	a	*b*

□

Intuitively, a constraint is satisfied if any of the relevant attributes has a *null* value or the constraint is satisfied in the traditional way, that is, as first-order satisfaction and with null values treated as any other constant.

In order to verify if an attribute takes the null value, the special predicate $\mathit{IsNull}(\cdot)$ is added to the language, with $\mathit{IsNull}(c)$ true iff c is *null*. This predicate is needed since using the built-in comparison atom $c = \mathit{null}$ will not work in traditional database management systems, where this equality would be always evaluated as *unknown* (the unique names assumption does not apply to null values [69]).

The semantics of constraint satisfaction in presence of null values is defined as follows.

Definition 2.5 [16] A constraint ψ of the form (2.2) is satisfied in the database instance D, denoted $D \models_N \psi$, iff $D^{\mathcal{A}(\psi)} \models \psi^N$, where ψ^N is:

$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i^{\mathcal{A}(\psi)}(\bar{x}_i) \ \rightarrow \ (\bigvee_{v_j \in \mathcal{A}(\psi)\cap\bar{x}} \mathit{IsNull}(v_j) \ \vee \ \exists \bar{z}(\bigvee_{j=1}^{n} Q_j^{\mathcal{A}(\psi)}(\bar{y}_j, \bar{z}_j) \ \vee \ \varphi))), \tag{2.5}$$

with $\bar{x} = \cup_{i=1}^{m} \bar{x}_i$ and $\bar{z} = \cup_{j=1}^{n} \bar{z}_j$. $D^{\mathcal{A}(\psi)} \models \psi^N$ refers to the classical first-order satisfaction with *null* treated as any other constant in $\mathcal{U}$. $\square$

In other words, an IC of the form (2.2) is satisfied (a) whenever there exists a *null* value in any of the relevant attributes in its antecedent, (b) if no null values appear in the antecedent, then the second disjunction in the consequent of formula (2.5) is satisfied, which correspond to the consequent of the original IC restricted to the relevant attributes. This check can be done as usual, treating nulls as any other domain constant.

Example 2.6 (example 2.5 cont.) To check if $D \models_N \psi$ with $\psi : \forall xyz(P(x,y,z) \rightarrow R(x,y))$, we need to verify if $D^{\mathcal{A}(\psi)} \models \forall xy(P^{\mathcal{A}(\psi)}(x,y) \rightarrow (\mathit{IsNull}(x) \vee \mathit{IsNull}(y) \vee R^{\mathcal{A}(\psi)}(x,y)))$. For $x = a$ and $y = b$, $D^{\mathcal{A}(\psi)} \models P^{\mathcal{A}(\psi)}(a,b)$, since none of them is a null value, i.e. $\mathit{IsNull}(a)$ and $\mathit{IsNull}(b)$ are both false, we need to check if $D^{\mathcal{A}(\psi)} \models R^{\mathcal{A}(\psi)}(a,b)$, which in this case is true.

For $x = b$ and $y = null$, $D^{\mathcal{A}(\psi)} \models P^{\mathcal{A}(\psi)}(b, null)$, since there is a null in a relevant attribute, i.e. *IsNull*$(null)$ is true, the constraint is trivially satisfied. As a consequence, and since there are no tuples that violate the IC, the database instance D is consistent regarding IC.

The database instance D' below is inconsistent regarding the UIC $\psi : \forall xyz\ (P(x, y, z) \rightarrow R(x, y))$.

P	X	Y	Z
	a	b	$null$
	b	b	a

R	X	Y
	a	b

This is because, for $x = b$ and $y = b$, $D^{\mathcal{A}(\psi)} \models P^{\mathcal{A}(\psi)}(b, b)$, but since none of them is a null value, we need to check if $D^{\mathcal{A}(\psi)} \models R^{\mathcal{A}(\psi)}(b, b)$, which in this case is false. $\square$

The predicate *IsNull* can be used to specify *NOT NULL*-constraints, which are common in commercial database management systems. A *NOT NULL*-constraint prevents certain attributes from taking a null value. As discussed before, this constraint is different from having $x \neq null$.

Definition 2.6 [16] A *NOT NULL*-constraint (NNC) is a denial constraint of the form

$$\bar{\forall}\bar{x}(P(\bar{x}) \wedge \mathit{IsNull}(x_i) \rightarrow \textbf{false}), \tag{2.6}$$

where $x_i \in \bar{x}$ is in the position of the attribute that cannot take null values. For a NNC ψ, $D \models_N \psi$ iff $D \models \psi$ in the classical sense, treating *null* as any other constant. $\square$

Notice that a NNC is not of the form (2.2), because it contains the special predicate *IsNull*. Nevertheless, when constructing the dependency graph for a set of ICs, NNCs will be treated as any unary IC, i.e. if there is a NNC on predicate P then there will exist an edge (P, P) from node P to P in the graph $\mathcal{G}(IC)$.

Example 2.7 Consider the database schema $\mathit{Student}(\mathit{ID}, \mathit{Name})$, the primary key of $\mathit{Student}$ can be specified by the UIC: $\forall xyz(\mathit{Student}(x, y) \wedge \mathit{Student}(x, z) \rightarrow y = z)$ together with the NNC $\forall xy(\mathit{Student}(x, y) \wedge \mathit{IsNull}(x) \rightarrow \mathbf{false})$. The UIC specifies that the first attribute of the relation is the primary key of it, and the NNC prevents that this attribute takes a null value.

The dependency graph for this set of ICs contains the edge $(\mathit{Student}, \mathit{Student})$. $\square$

2.3 Database Repairs and Repair Programs

When inconsistencies arise in a database instance D, consistency can be restored by deleting and/or inserting tuples. In particular, RICs (of the form (2.4)) are repaired by tuple deletions or tuple insertion with null values. In this way, a repair is a new database instance with the same schema as D that satisfies the ICs and differs minimally (under set inclusion) from the D [2].

In order to formally define database repairs, we need to introduce the following concepts:

Definition 2.7 [2] Let D, D' be database instances over the same schema and domain. The *distance*, $\Delta(D,\ D')$, between D and D' is the symmetric difference $\Delta(D, D') = (D \smallsetminus D') \cup (D' \smallsetminus D)$. $\square$

It is possible to define a partial order between database instances.

Definition 2.8 [16] Let D, D', D'' be database instances over the same schema and domain $\mathcal{U}$. It holds $D' \leq_D D''$ iff:

1. For every atom $P(\bar{a}) \in \Delta(D, D')$, with $\bar{a} \in (\mathcal{U} \smallsetminus \{\mathit{null}\})$,[6] it holds that $P(\bar{a}) \in$

[6]that $\bar{a} \in (\mathcal{U} - \{\mathit{null}\})$ means that each of the elements in tuple $\bar{a}$ belongs to $(\mathcal{U} - \{\mathit{null}\})$.

$\Delta(D, D'')$.

2. For every atom $Q(\bar{a}, \overline{null}) \in \Delta(D, D')$,[7] with $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})$, there exists a $\bar{b} \in \mathcal{U}$ such that $Q(\bar{a}, \bar{b}) \in \Delta(D, D'')$ and $Q(\bar{a}, \bar{b}) \notin \Delta(D, D')$. $\square$

This partial order is used to define the *repairs* of an inconsistent database.

Definition 2.9 [16] Given a database instance D and a set IC of ICs of the form (2.2), and NNCs of the form (2.6), a repair of D wrt IC is a database instance D' over the same schema of D, such that:

(a) $D' \models_N IC$.

(b) D' is $\leq_D$-*minimal* in the class of database instances that satisfy IC wrt $\models_N$, i.e. there is no database D'' in this class with $D'' <_D D'$, where $D'' <_D D'$ means $D'' \leq_D D'$ but not $D' \leq_D D''$.

The set of repairs of D wrt IC is denoted by $Rep(D, IC)$. $\square$

In the absence of *null*, this definition of repair coincides with the one in [2].

We assume that our set IC of ICs, consisting of ICs of the form (2.2) and NNCs of the form (2.6) are *non-conflicting*, in the sense that there is no NNC on an attribute of a relation that is existentially quantified in an IC of the form (2.2).

Example 2.8 The database instance $D = \{S(a), S(b), R(b)\}$ is inconsistent wrt IC: $\forall x(S(x) \rightarrow R(x))$, since $S(a)$ is in D, but $R(a)$ is not. Consistency can be restored by inserting $R(a)$ or deleting $S(a)$. Table 2.1 shows the two database repairs of D and the difference, in terms of whole tuples, wrt the original database instance D.

[7] $\overline{null}$ is a tuple of null values, that for simplification purposes, are placed in the last attributes of Q, but could be anywhere else in Q.

i	D_i	$\Delta(D, D_i)$
1	$\{S(a), S(b), R(a), R(b)\}$	$\{R(a)\}$
2	$\{S(b), R(b)\}$	$\{S(a)\}$

Table 2.1: Database Repairs

The database instance $D_3 = \{\}$ is consistent wrt IC, but it is not a repair since it does not satisfy minimality. In fact, $\Delta(D, D_3) = \{S(a), S(b), R(b)\}$, and $D_2 <_D D_3$.

For database instance $D = \{P(a, null), P(b, c), R(a, b)\}$ and IC: $\forall xy(P(x, y) \rightarrow \exists z R(x, z))$, there are two repairs: $D_1 = \{P(a, null), P(b, c), R(a, b), R(b, null)\}$, with $\Delta(D, D_1) = \{R(b, null)\}$, and $D_2 = \{P(a, null), R(a, b)\}$, with $\Delta(D, D_2) = \{P(b, c)\}$. The database instance $D_3 = \{P(a, null), P(b, c), R(a, b), R(b, d)\}$, for any $d \in \mathcal{U}$ different from $null$, is not a repair: Since $\Delta(D, D_3) = \{R(b, d)\}$, we have $D_1 <_D D_3$ and, therefore D_3 is not $\leq_D$-minimal. $\square$

Database repairs can be specified as stable models (*SM*) of *disjunctive logic programs* [44, 68]. The idea is that, given an inconsistent database instance D and a set IC of RIC-acyclic ICs, a disjunctive repair program $\Pi(D, IC)$ is constructed, such that there is a one-to-one correspondence between the stable models of $\Pi(D, IC)$ and the repairs of D [7, 16].

Annotation	Atom	The tuple $P(\bar{a})$ is ...
$\mathbf{t_d}$	$P(\bar{a}, \mathbf{t_d})$	$P(\bar{a})$ is true in the database.
$\mathbf{t_a}$	$P(\bar{a}, \mathbf{t_a})$	$P(\bar{a})$ is advised to be made true.
$\mathbf{f_a}$	$P(\bar{a}, \mathbf{f_a})$	$P(\bar{a})$ is advised to be made false.
$\mathbf{t^\star}$	$P(\bar{a}, \mathbf{t^\star})$	$P(\bar{a})$ is true or is made true.
$\mathbf{t^{\star\star}}$	$P(\bar{a}, \mathbf{t^{\star\star}})$	$P(\bar{a})$ is true in *the repair*.

Table 2.2: Annotation Constants

As mentioned before, repair programs use annotation constants, whose role is to enable the definition of atoms that can become true in the repairs or false in order to satisfy the ICs. The idea is to use logic rules to specify how a database violates certain ICs, and how the database can become consistent wrt the ICs. Actually, each atom of the form $P(\bar{a})$ can receive one of the constants in Table 2.2.

Annotations are performed according to the following sequential steps: first ground atoms $P(\bar{c})$ from the database receive an extra argument $\mathbf{t_d}$, as a consequence, $P(\bar{a}, \mathbf{t_d})$ becomes a fact in $\Pi(D, IC)$. Next, for each IC a disjunctive rule is constructed in such a way that the body of the rule captures the violation condition for the IC; and the head describes how to restore the consistency by deleting or inserting the corresponding tuples. These endorsements are seized by the $\mathbf{t_a}, \mathbf{f_a}$ annotations. For instance, atom $P(\bar{a}, \mathbf{t_a})$ establishes the insertion of $P(\bar{a})$; and $P(\bar{a}, \mathbf{f_a})$, the deletion of $P(\bar{a})$. As an illustration, for the inclusion dependency $\forall x(S(x) \rightarrow R(x))$, the disjunctive program rule:

$$S(x, \mathbf{f_a}) \vee R(x, \mathbf{t_a}) \leftarrow S(x, \mathbf{t_d}),\ \mathit{not}\ R(x, \mathbf{t_d}), \tag{2.7}$$

states that if the tuple $S(x, \mathbf{t_d})$ is a program fact, but $R(x, \mathbf{t_d})$ is not, then consistency is restored by deleting $S(x)$, which receives constant $\mathbf{f_a}$ in the head of the rule, or by inserting $R(x)$, which receives the $\mathbf{t_a}$ constant.

The $\mathbf{t}^\star$ constant is introduced in order to keep repairing the database if there is interaction of ICs. Thus, it becomes highly significant in cases where the insertion of a tuple may generate a new IC violation, e.g. if due to a different IC, $S(c, \mathbf{t_a})$ is generated and $R(c)$ is not in the database, or $R(c)$ has been made false, the constraint is violated once again. The aftermath is that the program rule (2.7) has to be changed to:

$$S(x, \mathbf{f_a}) \vee R(x, \mathbf{t_a}) \leftarrow S(x, \mathbf{t}^\star),\ \mathit{not}\ R(x, \mathbf{t_d}), \tag{2.8}$$

where the atom $S(x, \mathbf{t}^\star)$ becomes true if either $S(x, \mathbf{t_d})$ or $S(x, \mathbf{t_a})$ are true. Moreover, rule (2.9) has to be added to the program. This rule captures the case when $S(c, \mathbf{t_a})$ has been generated, but $R(c)$ has been made false, which again causes the violation of the IC.

$$S(x, \mathbf{f_a}) \vee R(x, \mathbf{t_a}) \leftarrow S(x, \mathbf{t}^\star), R(x, \mathbf{f_a}), \tag{2.9}$$

Finally, atoms with constant $\mathbf{t}^{\star\star}$ are the ones that become true in the repairs. They are use to read off the database atoms in the repairs. The following program was introduced in [7, 12].

Definition 2.10 [7, 12] The repair program $\Pi(D, IC)$ for a database instance D and set IC of UICs, RICs and NNCs is composed by the following rules:

1. $dom(a)$, for each constant $a \in (\mathcal{U} - \{null\})$.

2. $P(\bar{a}, \mathbf{t_d})$, for each atom $P(\bar{a}) \in D$.

3. For every UIC ψ of the form (2.3), the set of rules:
   $\bigvee_{i=1}^n P_i(\bar{x}_i, \mathbf{f_a}) \vee \bigvee_{j=1}^m Q_j(\bar{y}_j, \mathbf{t_a}) \ \leftarrow \ \bigwedge_{i=1}^n P_i(\bar{x}_i, \mathbf{t}^\star), \bigwedge_{Q_j \in Q'} Q_j(\bar{y}_j, \mathbf{f_a}),$
   $\bigwedge_{Q_k \in Q''} \ not\ Q_k(\bar{y}_k, \mathbf{t_d}), \bigwedge_{x_l \in (\mathcal{A}(\psi) \cap \bar{x})} dom(x_l), \bar{\varphi},$
   for every set Q' and Q'' of atoms appearing in formula (2.3) such that $Q' \cup Q'' = \bigcup_{i=1}^m Q_i$ and $Q' \cap Q'' = \emptyset$, where $\mathcal{A}(\psi)$ is the set of relevant attributes for ψ, $\bar{x} = \bigcup_{i=1}^n x_i$, and $\bar{\varphi}$ is a conjunction of built-ins that is equivalent to the negation of φ.

4. For every RIC ψ of the form (2.4), the rules:
   $P(\bar{x}, \mathbf{f_a}) \vee Q(\bar{y}, \overline{null}, \mathbf{t_a}) \leftarrow P(\bar{x}, \mathbf{t}^\star),\ not\ aux_\psi(\bar{y}),\ dom(\bar{y}).$
   And for every $z_i \in \bar{z}$:
   $aux_\psi(\bar{y}) \leftarrow Q(\bar{y}, \bar{z}, \mathbf{t}^\star),\ \ not\ Q(\bar{y}, \bar{z}, \mathbf{f_a}),\ dom(\bar{y}),\ dom(z_i).$

5. For every NNC of the form (2.6), the rule:
   $P(\bar{x}, \mathbf{f_a}) \leftarrow P(\bar{x}, \mathbf{t}^\star), x_i = null.$

6. For each predicate $P \in R$, the annotation rules:
   $P(\bar{x}, \mathbf{t}^\star) \leftarrow P(\bar{x}, \mathbf{t_d}). \quad P(\bar{x}, \mathbf{t}^\star) \leftarrow P(\bar{x}, \mathbf{t_a}).$

7. For every predicate $P \in \mathcal{R}$, the interpretation rules:

   $P(\bar{x}, \mathbf{t^{\star\star}}) \leftarrow P(\bar{x}, \mathbf{t_a}). \quad P(\bar{x}, \mathbf{t^{\star\star}}) \leftarrow P(\bar{x}, \mathbf{t_d}),\ not\ P(\bar{x}, \mathbf{f_a}).$

8. For every predicate $P \in \mathcal{R}$, the program denial constraint:

   $\leftarrow P(\bar{x}, \mathbf{t_a}), P(\bar{x}, \mathbf{f_a}).$ $\square$

The rules in 1. capture the database constants except for the *null*, which are stored in an auxiliary predicate *dom*. The rules in 2. establish the program facts which are the elements of the database. The rules in 3., 4., and 5. capture in the right-hand side the violation of ICs of the form (2.3), (2.4), and (2.6), respectively, and with the left-hand side the intended way of restoring consistency.

In particular, the set of predicates Q' and Q'' in rules in 3. are used to check that in all the possible combinations, the consequent of an UIC is not being satisfied. The rules in 4. enforce the satisfaction of a RIC, for instance if $P(\bar{a}, \mathbf{t^\star})$ is true and $aux(\bar{a})$ is false, i.e. there is no $\bar{z}$ such that $Q(\bar{a}, \bar{z})$ is true or was made true by the repair program, then there consistency can be restored by deleting $P(\bar{a})$ or by adding $Q(\bar{a}, null)$ to the database. Notice that the *aux* predicate permits to check the existence of such $Q(\bar{a}, \bar{z})$ atom.

Moreover, since the satisfaction of UICs and RICs needs to be checked only if none of the relevant attributes of the antecedent are *null*, we use $dom(x)$ in rules in 3., and in the two rules in 4. $dom(\bar{y})$ denotes the conjunction of the atoms $dom(y_j)$ for $y_j \in \bar{y}$. Notice that rules in 4. are implicitly based on the fact that the relevant attributes for a RIC of the form (2.4) are $\mathcal{A} = \{y \mid y \in \bar{y}\}$.

The rules in 6. capture the atoms that become true in the program. The rules in 7. capture the atoms that become true in the repairs. The rule in 8. represents the program denial constraints, i.e. the rules that discard the models where a same tuple is both deleted and inserted.

Program constraints are head-free rules; program denial constraints are program constraints with only positive and built-in atoms in the body. (Database) denial constraints are ICs, i.e. conditions that have to be satisfied by the database relations, that can be written as program denial constraints. However the role of a program constraint (denial or not) is to discard the stable models that violate them. In the following we will use "(denial) constraint" for the database case, and "program (denial) constraint" for programs.

Database repairs are retrieved from the stable models of $\Pi(D, IC)$: for each stable model $\mathcal{M}$ of $\Pi(D, IC)$, a repair is generated by selecting the atoms with $\mathbf{t}^{\star\star}$ constant in $\mathcal{M}$.

Example 2.9 The repair program $\Pi(D, IC)$ for $D = \{P(a, b, \mathit{null}), P(b, b, a), R(a, b), S(a, b, b), S(\mathit{null}, b, b)\}$, and IC: $\forall xyz(P(x, y, z) \rightarrow R(x, y))$, $\forall xy(R(x, y) \rightarrow \exists z S(x, y, z))$, and $\forall xyz(S(x, y, z) \wedge \mathit{IsNull}(x) \rightarrow \mathbf{false})$ contains the following rules:

1. $\mathit{dom}(a).\ \ \mathit{dom}(b).$

2. $P(a, b, \mathit{null}, \mathbf{t_d}).\quad P(b, b, a, \mathbf{t_d}).\quad R(a, b, \mathbf{t_d}).\quad S(a, b, b, \mathbf{t_d}).\ \ S(\mathit{null}, b, b, \mathbf{t_d}).$

3. $P(x, y, z, \mathbf{f_a}) \vee R(x, y, \mathbf{t_a}) \leftarrow P(x, y, z, \mathbf{t}^\star), R(x, y, \mathbf{f_a}), \mathit{dom}(x), \mathit{dom}(y).$

   $P(x, y, z, \mathbf{f_a}) \vee R(x, y, \mathbf{t_a}) \leftarrow P(x, y, z, \mathbf{t}^\star),\ \ \mathit{not}\ R(x, y, \mathbf{t_d}), \mathit{dom}(x), \mathit{dom}(y).$

4. $R(x, y, \mathbf{f_a}) \vee S(x, y, \mathit{null}, \mathbf{t_a}) \leftarrow R(x, y, \mathbf{t}^\star),\ \ \mathit{not}\ \mathit{aux}(x, y), \mathit{dom}(x), \mathit{dom}(y).$

   $\mathit{aux}(x, y) \leftarrow S(x, y, z, \mathbf{t}^\star),\ \ \mathit{not}\ S(x, y, z, \mathbf{f_a}), \mathit{dom}(x), \mathit{dom}(y), \mathit{dom}(z).$

5. $S(x, y, z, \mathbf{f_a}) \leftarrow S(x, y, z, \mathbf{t}^\star), x = \mathit{null}.$

6. $P(x, y, z, \mathbf{t}^\star) \leftarrow P(x, y, z, \mathbf{t_d}).$

   $P(x, y, z, \mathbf{t}^\star) \leftarrow P(x, y, z, \mathbf{t_a}).$

   $R(x, y, \mathbf{t}^\star) \leftarrow R(x, y, \mathbf{t_d}).$

   $R(x, y, \mathbf{t}^\star) \leftarrow R(x, y, \mathbf{t_a}).$

   $S(x, y, z, \mathbf{t}^\star) \leftarrow S(x, y, z, \mathbf{t_d}).$

   $S(x, y, z, \mathbf{t}^\star) \leftarrow S(x, y, z, \mathbf{t_a}).$

7. $P(x,y,z,\mathbf{t}^{\star\star}) \leftarrow P(x,y,z,\mathbf{t_a})$.

   $P(x,y,z,\mathbf{t}^{\star\star}) \leftarrow P(x,y,z,\mathbf{t_d}),\ \mathit{not}\ P(x,y,z,\mathbf{f_a})$.

   $R(x,y,\mathbf{t}^{\star\star}) \leftarrow R(x,y,\mathbf{t_a})$.

   $R(x,y,\mathbf{t}^{\star\star}) \leftarrow R(x,y,\mathbf{t_d}),\ \mathit{not}\ R(x,y,\mathbf{f_a})$.

   $S(x,y,z,\mathbf{t}^{\star\star}) \leftarrow S(x,y,z,\mathbf{t_a})$.

   $S(x,y,z,\mathbf{t}^{\star\star}) \leftarrow S(x,y,z,\mathbf{t_d}),\ \mathit{not}\ S(x,y,z,\mathbf{f_a})$.

8. $\leftarrow P(x,y,z,\mathbf{t_a}), P(x,y,z,\mathbf{f_a})$.

   $\leftarrow R(x,y,\mathbf{t_a}), R(x,y,\mathbf{f_a})$.

   $\leftarrow S(x,y,z,\mathbf{t_a}), S(x,y,z,\mathbf{f_a})$.

The rules in 3. establish the form of repairing the database according to the UIC, i.e. by making $P(x,y,z)$ false or $R(x,y)$ true. These rules are constructed by choosing all the possible sets Q' and Q'' such that $Q' \cup Q'' = \{R(x,y)\}$ and $Q' \cap Q'' = \emptyset$. The first rule in 3 considers $Q' = \{R(x,y)\}$ and $Q'' = \emptyset$. The second corresponds to $Q' = \emptyset$ and $Q'' = \{R(x,y)\}$. Note that *dom* atoms are only generated for the relevant attributes to check the UIC. The rules in 4. specify the form of restoring consistency wrt the RIC, i.e. by deleting $R(x,y)$ or inserting tuple $S(x,y,\mathit{null})$. *dom* atoms are only generated for the variables in the antecedent of the RIC. The rule 5 establishes the way of restoring consistency wrt the NNC, i.e. by eliminating atom $S(x,y,z)$. This program has two stable models:

$$\begin{aligned}\mathcal{M}_1 = \{&\mathit{dom}(a), \mathit{dom}(b), P(a,b,\mathit{null},\mathbf{t_d}), P(a,b,\mathit{null},\mathbf{t}^\star), P(b,b,a,\mathbf{t_d}), P(b,b,a,\mathbf{t}^\star),\\ &R(a,b,\mathbf{t_d}), R(a,b,\mathbf{t}^\star), S(a,b,b,\mathbf{t_d}), S(a,b,b,\mathbf{t}^\star), S(\mathit{null},b,b,\mathbf{t_d}), \mathit{aux}(a,b),\\ &S(\mathit{null},b,b,\mathbf{f_a}), S(\mathit{null},b,b,\mathbf{t}^\star), \underline{P(a,b,\mathit{null},\mathbf{t}^{\star\star})}, \underline{P(b,b,a,\mathbf{t}^{\star\star})}, \underline{R(a,b,\mathbf{t}^{\star\star})},\\ &R(b,b,\mathbf{t_a}), R(b,b,\mathbf{t}^\star), \underline{R(b,b,\mathbf{t}^{\star\star})}, S(b,b,\mathit{null},\mathbf{t_a}), S(b,b,\mathit{null},\mathbf{t}^\star),\\ &\underline{S(b,b,\mathit{null},\mathbf{t}^{\star\star})}, \underline{S(a,b,b,\mathbf{t}^{\star\star})}\},\end{aligned}$$

$$\begin{aligned}\mathcal{M}_2 \;=\; &\{dom(a), dom(b), P(a,b,null,\mathbf{t_d}), P(a,b,null,\mathbf{t^\star}), P(b,b,a,\mathbf{t_d}), P(b,b,a,\mathbf{t^\star}),\\ &R(a,b,\mathbf{t_d}), R(a,b,\mathbf{t^\star}), S(a,b,b,\mathbf{t_d}), S(a,b,b,\mathbf{t^\star}), S(null,b,b,\mathbf{t_d}), aux(a,b),\\ &S(null,b,b,\mathbf{f_a}), S(null,b,b,\mathbf{t^\star}), \underline{P(a,b,null,\mathbf{t^{\star\star}})}, P(b,b,a,\mathbf{f_a}), \underline{R(a,b,\mathbf{t^{\star\star}})},\\ &\underline{S(a,b,b,\mathbf{t^{\star\star}})}\}.\end{aligned}$$

Thus, consistency is recovered by inserting atoms $\{R(b,b), S(b,b,null)\}$ ($\{R(b,b,\mathbf{t_a}), S(b,b,null,\mathbf{t_a})\} \in \mathcal{M}_1$) and deleting atom $S(null,b,b)$ ($S(null,b,b,\mathbf{f_a}) \in \mathcal{M}_1$); or by deleting atoms $\{P(b,b,a), S(null,b,b)\}$ ($\{P(b,b,a,\mathbf{f_a}), S(null,b,b,\mathbf{f_a})\} \in \mathcal{M}_2$). The repairs are $\{P(a,b,null), R(a,b), S(a,b,b), P(b,b,a), R(b,b), S(b,b,null)\}$ and $\{P(a, b, null), R(a,b), S(a,b,b)\}$, as expected. $\square$

It was proved in [7, 16] that the repair program of Definition 2.10 is a correct specification of database repairs wrt *RIC-acyclic* sets of UICs of the form (2.3), RICs of the form (2.4), and NNCs of the form (2.6).

There are different notions of minimality of database repairs in the literature. For instance, in [3], minimality is based in cardinality of the set of changes. In [14, 38, 76] minimality is based in cardinality of the set of updates, i.e. changes of attributes values as opposed to whole tuples. Also, there are different repair policies in the literature. As an illustration, in [27] the database instance is assumed to be possibly incorrect but complete, then repairs are obtained by deletion of tuples only, i.e. the insertion of new tuples is not consider as an option to restore consistency. In [17] the database instance is assumed to be possibly incorrect and incomplete, then functional dependencies are repaired by deletion, and referential ICs by adding arbitrary elements of the domain. These and other alternative policies can be specified by repair programs.

2.4 Consistent Query Answering

First-order queries are formulas over the same first-order language $\mathcal{L}$ of the integrity constraints.

Definition 2.11 Given a database instance D, a tuple of constants $\bar{t}$ is an answer to a query $\mathcal{Q}(\bar{x})$ in D iff $D \models \mathcal{Q}(\bar{t})$, i.e. $\mathcal{Q}(\bar{x})$ becomes true in D when the variables are replaced by the corresponding constants in $\bar{t}$. $\Box$

A consistent answer to a FO query posed to a possibly inconsistent database D wrt a set IC of ICs is defined as follows:

Definition 2.12 [2] Given a database instance D, a tuple $\bar{t}$ is a consistent answer to a query $\mathcal{Q}(\bar{x})$ in D iff $\bar{t}$ is an answer to query $\mathcal{Q}(\bar{x})$ in every repair D' of D. Moreover, if a query $\mathcal{Q}$ is an $\mathcal{L}$-sentence, i.e. a boolean query, the consistent answer is *yes* if $\mathcal{Q}$ is true in every repair D' of D; and *no*, otherwise. The set of consistent answers to a query $\mathcal{Q}$ in D wrt IC is denoted by $ConsA(\mathcal{Q})$. $\Box$

In order to use repair programs to compute consistent answers, first-order queries posed over inconsistent databases are translated into logic programs. Given a query $\mathcal{Q}$, a new query $\Pi(\mathcal{Q})$ is generated by first expressing it as a Datalog program [64], and next replacing every positive literal $P(\bar{s})$ by $P(\bar{s}, \mathbf{t}^{\star\star})$, and every negative literal *not* $P(\bar{s})$ by *not* $P(\bar{s}, \mathbf{t}^{\star\star})$. Thus, in order to get consistent answers, $\Pi(\mathcal{Q})$ is "run" together with the corresponding repair program $\Pi(D, IC)$. As a consequence, consistent query answering is translated into cautious reasoning under the stable models semantics [44, 68].

Example 2.10 (example 2.9 cont.) For the Datalog query $\mathcal{Q}$: $Ans(x,y) \leftarrow R(x,y)$, $\Pi(\mathcal{Q})$ is $Ans(x,y) \leftarrow R(x,y,\mathbf{t^{\star\star}})$. There are two stable models of $\Pi(D,IC) \cup \Pi(\mathcal{Q})$:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{dom(a), dom(b), P(a,b,null,\mathbf{t_d}), P(a,b,null,\mathbf{t^\star}), P(b,b,a,\mathbf{t_d}), P(b,b,a,\mathbf{t^\star}),\\
& R(a,b,\mathbf{t_d}), R(a,b,\mathbf{t^\star}), S(a,b,b,\mathbf{t_d}), S(a,b,b,\mathbf{t^\star}), S(null,b,b,\mathbf{t_d}), aux(a,b),\\
& S(null,b,b,\mathbf{f_a}), S(null,b,b,\mathbf{t^\star}), P(a,b,null,\mathbf{t^{\star\star}}), P(b,b,a,\mathbf{t^{\star\star}}), R(a,b,\mathbf{t^{\star\star}}),\\
& R(b,b,\mathbf{t_a}), R(b,b,\mathbf{t^\star}), R(b,b,\mathbf{t^{\star\star}}), S(b,b,null,\mathbf{t_a}), S(b,b,null,\mathbf{t^\star}),\\
& S(b,b,null,\mathbf{t^{\star\star}}), S(a,b,b,\mathbf{t^{\star\star}}), \underline{Ans(a,b)}, \underline{Ans(b,b)}\},\\
\mathcal{M}_2 \;=\; & \{dom(a), dom(b), P(a,b,null,\mathbf{t_d}), P(a,b,null,\mathbf{t^\star}), P(b,b,a,\mathbf{t_d}), P(b,b,a,\mathbf{t^\star}),\\
& R(a,b,\mathbf{t_d}), R(a,b,\mathbf{t^\star}), S(a,b,b,\mathbf{t_d}), S(a,b,b,\mathbf{t^\star}), S(null,b,b,\mathbf{t_d}), aux(a,b),\\
& S(null,b,b,\mathbf{f_a}), S(null,b,b,\mathbf{t^\star}), P(a,b,null,\mathbf{t^{\star\star}}), P(b,b,a,\mathbf{f_a}), R(a,b,\mathbf{t^{\star\star}}),\\
& S(a,b,b,\mathbf{t^{\star\star}}), \underline{Ans(a,b)}\}.
\end{aligned}$$

The only *Ans*-atom in common is $Ans(a,b)$, therefore $ConsA(\mathcal{Q}) = (a,b)$. □

In the general case, CQA over inconsistent databases is an expensive computational task. In fact, its worst case data complexity is similar to the complexity of cautious reasoning under stable models semantics, i.e. Π_2^P-complete [30]. Nevertheless, it is possible to identify classes of ICs and queries for which data complexity is lower than the worst-case data complexity. Some polynomial cases of CQA have been reported in [2, 7, 27, 39].

Chapter 3

Thesis Contributions

3.1 Statement of the Problem and Objectives

Since, in the general case, CQA over inconsistent databases is as expensive as the evaluation of disjunctive logic programs under the stable models semantics (in data complexity) [28, 30], we need the expressive language of disjunctive logic programs to handle CQA. Nevertheless, using logic programs in a straightforward way is usually inefficient. Therefore, it becomes relevant to optimize logic programs and query evaluation from them.

In this thesis, we are interested in optimizing and implementing the general logic programming approach to CQA for stand alone databases. Essentially, we are concerned with improving the structure of repair programs, model computation, and query evaluation from them. Structural optimizations of programs involve changing the program without affecting the repair semantics. This implies the elimination of redundant rules, auxiliary predicates, and (some) annotation constants.

With respect to improving the evaluation of logic programs, there are two important issues to consider: First, consistent answers are obtained from stable models for the combination of the repair and query programs. But, in most of the cases only a subset of the program and the database facts is needed to compute answers to a specific query. We explore the use of magic sets (MS) methodologies [8] to capture that subset. MS optimizes the bottom-up processing of queries by simulating a top-down evaluation of queries [23], which permits to focalize on a part of the program and base data, instead of considering the whole sets of rules and facts. Actually, only the rules and database facts that involve predicates and parameters related to the predicates and values in the query are taken into account. In particular, with MS only a relevant subset of the database is used for query evaluation.

Second, we develop a method that reduces the flow of data between the database system and the reasoning system. This method selects and imports only the relevant base data to compute queries from the database system, where the data resides, into the reasoning system where programs are evaluated.

We implement optimized logic programs and methods to compute consistent answers from them. The *optimized system*, which is based on logic programming, retrieves consistent answers to queries from stand alone relational databases. This is very relevant, because as far as we know, there is no other implementation of CQA for universal and referential ICs, and general first-order queries. Our system implements the semantics of constraints satisfaction defined in [16], and presented in Chapter 2, which works for databases that may contain null values.

Furthermore, it is possible to identify classes of ICs and queries for which CQA has lower data complexity. For example, CQA regarding sets of universal ICs and projection free conjunctive queries is polynomial in data complexity [2]. In [7], head-cycle free disjunctive repair programs are detected and translated into equivalent normal programs with lower computational complexity (*coNP*-complete) [30]. Other lower complexity classes for CQA are identified in [27, 39]. For all those cases, alternative methods for CQA can be implemented.

In this direction, we show that there are classes of ICs and queries for which we can compute consistent answers by using a *core computation* as an alternative to computing and querying all the stable models of the repair program and query program. The *core* of the original database (or of the repair program) wrt a set of

ICs is the set of database atoms in the intersection of all its repairs, or equivalently, of database atoms in the intersection of all stable models of the repair program. The core can be captured by the *well-founded semantics* of the program, in which case the core can be computed in polynomial time [60]. Core computations have been considered before for CQA for aggregate queries [4].

The *well-founded semantics* for normal logic programs was introduced in [75], and later extended to disjunctive logic programs [60, 67]. It has been used as an alternative to the stable models semantics [43, 44, 68]. In this thesis, we show that under certain conditions, for UICs and RICs, and conjunctive queries without existential quantifiers, the intersection of the stable models of the program composed by the repair and query program, coincides with the set of true atoms in the well-founded interpretation (WFI) of that program, generalizing some preliminary results obtained in [3] (for a different kind of repair programs). This property is significant, because in those cases CQA becomes polynomial in data complexity.

Additionally, we analyze the use of the WFS as a first step towards answering ground disjunctive queries, leaving the stable models semantics for a second stage, only if necessary. We also consider the use of the WFS as a general way of computing consistent answers, and by doing so and by complexity theoretic reasons, just providing a lower complexity *approximation* to CQA. As an illustration, with the WFS we retrieve a subset of the consistent answers to positive Datalog queries wrt RIC-acyclic sets of ICs (cf. Definition 2.2).

Moreover, we extend logic programs to compute consistent answers to aggregate queries with *scalar* functions, and aggregate queries with *group-by* statements. Our motivation is to use both, the logic programs to specify database repairs, and the capabilities of DLV system to compute aggregates over stable models. As we mentioned before, the former queries apply one of the aggregate functions *min, max, count, sum,* *avg* over an attribute of a database relation, and return a unique value for the whole relation. On the opposite, queries with *group-by* statements perform grouping on the values of an attribute or a set of attributes, and apply one of the aggregate functions over each group. The semantics for consistent answers to *scalar* aggregate queries was given in [4]. The semantics for consistent answers to aggregate queries with *group-by* statements is presented in this thesis. By using repair programs, we exploit the aggregation capabilities of current reasoning systems to compute aggregate functions over stable models, such as the DLV system [61], a state of the art system for disjunctive logic programming.

Finally, CQA has been mostly analyzed in relational databases and in data integration systems, but there is no literature on CQA for Multidimensional Databases (MDB). For this reason, we develop a semantic framework for CQA for Multidimensional Databases [48]. We focus on Multidimensional Data Warehouses (MDWs), which are data repositories that integrate data from different sources, and keep historical data [24]. Basically, MDWs consist mainly of dimensions and facts. The former reflect the way in which the data is organized, e.g. time, location, customers, etc. The latter correspond to quantitative data related to the dimensions, e.g. facts related with sales may be associated to the dimensions time and location, and should be understood as the sales at the locations in certain periods of time.

In a multidimensional data model [48], dimensions are represented by hierarchical schemas together with a set of dimension constraints, while the facts are represented by tables that refer to the dimensions. Dimensions are considered as the static part of the data warehouses, whereas the facts are considered as the dynamic part, so that the update operations affect mainly the fact tables. However, in [50, 51] it was shown that dimensions can be updated; dimensions constraints can be violated; and therefore, MDWs may become inconsistent wrt them, affecting the evaluation of queries. For MDBs a new repair semantics is required, since the relational notion of database repair presented in [2] cannot be applied to MDBs, mainly because of the different data schemas and ICs. This is presented in Chapter 8.

3.2 Overview of Results

Even though we cannot reduce the intrinsic complexity of CQA over inconsistent databases, we can optimize the computation of consistent answers from inconsistent databases. The contributions of this dissertation can be summarized as follows:

1. A simplified version of the repair programs of Definition 2.10. Essentially, the annotations on database facts and auxiliary predicates are eliminated. Moreover, we make an intelligent generation of program denial constraints, so that they are generated only when needed. The program denial constraints are the rules that avoid that atoms becomes simultaneously annotated with both $\mathbf{t_a}$ and $\mathbf{f_a}$ constants.

2. A suitable Magic Sets methodology for disjunctive repair programs. Magic sets techniques allow to focalize on parts of the repair programs and facts that are relevant to answer a query. We prove that our magic sets methodology is sound and complete when it is applied to disjunctive repair programs with program denial constraints.

3. A methodology to import into a reasoning system the relevant base data to compute queries from a database instance.

4. The development, implementation, and description of the "Consistency Extractor System", an optimized logic programming-based implementation to compute consistent answers to FO queries from stand alone relational databases.

5. An analysis of the use of the well-founded semantics for CQA. We identify classes of ICs and queries for which the well-founded semantics of programs [60, 67] provides the same consistent answers to queries as the stable models semantics [44, 68].

6. A logic programming specification of database repairs to compute consistent answers to aggregate queries with both *scalar* functions and *group-by* statements. We also provide a guide to compute consistent answers to aggregate queries in the DLV reasoning system [61].

7. A repair semantics for Multidimensional Databases. We define a suitable notion of repair for multidimensional dimension instances.

Chapter 4

Structural Optimizations of Repair Programs

4.1 Introduction

In this chapter we describe structural optimizations to the logic programs in Definition 2.10.

Basically, structural modifications involve changing the program without affecting the repair semantics. Thus, we eliminate redundant rules, auxiliary predicates, and some annotation constants. In addition, we make an intelligent generation of program constraints, so that they are generated only when needed. Moreover, classes of ICs are identified, for which repair programs do not contain program constraints at all. This is important because, apart of eliminating unnecessary model checking, it allows for the application of magic sets as implemented in the DLV system (cf. Chapter 5), which currently requires the absence of program constraints. It has been shown that magic sets considerably improve the evaluation of queries [29].

Through structural optimizations we obtain simpler repair programs, which are easier to evaluate by a reasoning system.

The remain of the chapter is structured as follows. Section 4.2 presents the changes on annotations, predicates and rules. Section 4.3 describes a method to generate relevant program constrains for repair programs. Section 4.4 presents the new optimized repair programs. Section 4.5 finalizes this chapter.

4.2 Auxiliary Annotations, Predicates, and Redundant Rules

The construction of repair programs and their evaluation can be improved by applying suitable structural modifications. For instance, in order to generate the program facts and domain constants, it is necessary to process the whole database, because facts need to be annotated. This technically means bringing the database into main memory. Therefore, given a large data set the construction of programs may become a slow process.

First, instead of manually inserting database facts into repair programs once annotated with the $\mathbf{t_d}$ constant, the facts are imported directly from the database without any annotation; and that constant is eliminated from the programs. In consequence, the database predicate P and its version that becomes expanded with an extra argument for the annotations have to be told apart. Therefore, the expanded version of a predicate P is replaced by an underscored version of the predicate, e.g. $P(\bar{a}, \mathbf{t_a})$ becomes $P\_(\bar{a}, \mathbf{t_a})$.

Second, the auxiliary *dom* predicate is also eliminated. That predicate was introduced to extract database constants, which are useful to check satisfiability of ICs. Thus, instead of checking that variables are restricted to the database domain, we check that variables do not take *null* values. This is achieved by adding in rules regarding ICs conditions of the form $\bar{x} \neq \mathit{null}$, instead of using $\mathit{dom}(\bar{x})$.

Finally, instead of having two interpretation rules for each database predicate, only one rule is used (cf. rules 7 in Definition 2.10). As previously introduced, the interpretation rules are:

$$P\_(\bar{x}, \mathbf{t}^{\star\star}) \leftarrow P\_(\bar{x}, \mathbf{t_a}). \tag{4.1}$$

$$P\_(\bar{x}, \mathbf{t}^{\star\star}) \leftarrow P(\bar{x}),\ \mathit{not}\ P\_(\bar{x}, \mathbf{f_a}). \tag{4.2}$$

These rules define the atoms that become true in the repairs; which are the ones advised to be true (Rule 4.1) or original database facts that are not advised to be false (Rule 4.2). Now, for each database predicate there is a single interpretation rule, namely:

$$P\_(\bar{x}, \mathbf{t}^{\star\star}) \leftarrow P\_(\bar{x}, \mathbf{t}^{\star}),\ \ not\ P\_(\bar{x}, \mathbf{f_a}). \tag{4.3}$$

With these modifications database facts do not have to be preprocessed by adding annotations to them, and the number of rules in the repair programs decreases. Notice that since the *dom* atoms were defined for each database constant in the database domain, the elimination of the *dom* rules becomes relevant for large databases.

4.3 Relevant Program Constraints

Program constraints of repair programs permit to discard incoherent models, i.e. models containing atoms annotated with both $\mathbf{t_a}$ and $\mathbf{f_a}$. We can identify cases of ICs for which a repair program will never generate such models. In those cases, program constraints can be eliminated. Apart from eliminating unnecessary model checking, the elimination of program constraints allows for the application of magic sets as implemented in the DLV system (cf. Chapter 5).

It can be seen that a repair program will have rules defining $P\_(\bar{x}, \mathbf{t_a})$, and $P\_(\bar{x}, \mathbf{f_a})$, for an atom $P(\bar{x})$ only if there exists at least two different ICs having $P(\bar{x})$ both in the antecedent of an IC and in the consequent of another IC. In those cases, program constraint for P should be kept.

Example 4.1 Given the database instance $D = \{S(a)\}$, and set IC: $\forall x(S(x) \rightarrow Q(x))$, $\forall x(Q(x) \rightarrow R(x))$, $\Pi(D, IC)$ has the following rules: $S(a)$.

$$\left.\begin{array}{l} S\_(x,\mathbf{t^\star}) \leftarrow S\_(x,\mathbf{t_a}).\\ S\_(x,\mathbf{t^\star}) \leftarrow S(x).\\ S\_(x,\mathbf{t^{\star\star}}) \leftarrow S\_(x,\mathbf{t^\star}),\ \textit{not}\ S\_(x,\mathbf{f_a}). \end{array}\right\} \text{(Similarly for } Q \text{ and } R\text{)}$$

$S\_(x,\mathbf{f_a}) \vee Q\_(x,\mathbf{t_a}) \leftarrow S\_(x,\mathbf{t^\star}), Q\_(x,\mathbf{f_a}), x \neq \textit{null}.$

$S\_(x,\mathbf{f_a}) \vee Q\_(x,\mathbf{t_a}) \leftarrow S\_(x,\mathbf{t^\star}),\ \textit{not}\ Q(x), x \neq \textit{null}.$

$Q\_(x,\mathbf{f_a}) \vee R\_(x,\mathbf{t_a}) \leftarrow Q\_(x,\mathbf{t^\star}), R\_(x,\mathbf{f_a}), x \neq \textit{null}.$

$Q\_(x,\mathbf{f_a}) \vee R\_(x,\mathbf{t_a}) \leftarrow Q\_(x,\mathbf{t^\star}),\ \textit{not}\ R(x), x \neq \textit{null}.$

$\leftarrow Q\_(x,\mathbf{t_a}), Q\_(x,\mathbf{f_a}).\qquad \leftarrow S\_(x,\mathbf{t_a}), S\_(x,\mathbf{f_a}).\qquad \leftarrow R\_(x,\mathbf{t_a}), R\_(x,\mathbf{f_a}).$

In the case of predicate S (and R) there is no way to generate an atom with constant $\mathbf{t_a}$ ($\mathbf{f_a}$ for R). Thus, the program constraints for S and R are always satisfied, and therefore, they can be eliminated. In contrast, for predicate Q both annotations are needed in the program, hence its program constraint has to be kept; otherwise we get incoherent stable models. The stable models of program $\Pi(D, IC)$ without the program constraints are:

$$\begin{aligned} \mathcal{M}_1 &= \{S(a), S\_(a,\mathbf{t^\star}), S\_(a,\mathbf{t^{\star\star}}), Q\_(a,\mathbf{t_a}), Q\_(a,\mathbf{t^\star}), Q\_(a,\mathbf{t^{\star\star}}), R\_(a,\mathbf{t_a}), R\_(a,\mathbf{t^\star}),\\ &\qquad R\_(a,\mathbf{t^{\star\star}})\},\\ \mathcal{M}_2 &= \{S(a), S\_(a,\mathbf{t^\star}), S\_(a,\mathbf{t^{\star\star}}), \underline{Q\_(a,\mathbf{t_a})}, Q\_(a,\mathbf{t^\star}), \underline{Q\_(a,\mathbf{f_a})}\},\\ \mathcal{M}_3 &= \{S(a), S\_(a,\mathbf{t^\star}), S\_(a,\mathbf{f_a})\}. \end{aligned}$$

$\mathcal{M}_2$ is an incoherent stable model and, therefore it cannot be considered as a model of program $\Pi(D, IC)$, since what we obtain from it is not a database repair. Actually, the database repairs are: $\{S(a), Q(a), R(a)\}$, and $\{\}$, which can be generated from stable models $\mathcal{M}_1$ ($\{Q\_(a,\mathbf{t_a}), R\_(a,\mathbf{t_a})\} \in \mathcal{M}_1$) and $\mathcal{M}_3$ ($S\_(a,\mathbf{f_a}) \in \mathcal{M}_3$). □

This idea can be formalized by appealing to the interaction between predicates as involved in ICs, which is captured by the dependency graph of Definition 2.1.

Example 4.2 (example 4.1 cont.) Figure 4.1 shows the dependency graph $\mathcal{G}(IC)$ for the set IC: $\forall x(S(x) \rightarrow Q(x))$, and $\forall x(Q(x) \rightarrow R(x))$.

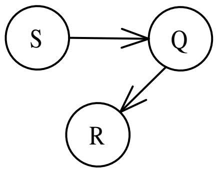


Figure 4.1: Dependency Graph $\mathcal{G}(IC)$ for a Set of UICs

Node S is connected to Q due to the first IC, Q is connected to R due to the second IC. S is a source node, R is a sink node. $\square$

Definition 4.1 Given a database instance D, and a set IC of ICs, program $\Pi'(D, IC)$ can be obtained from $\Pi(D, IC)$ by deleting program constraints for the predicates that are *sinks* or *sources* in the dependency graph $\mathcal{G}(IC)$. $\square$

Example 4.3 (example 4.1 and 4.2 cont.) Program $\Pi'(D, IC)$ has the same set of rules as program $\Pi(D, IC)$, except for the program constraints for predicates S and R. This happens because, since S is a source node and R is sink in the dependency graph in Figure 4.1, program constraints related to them can be deleted. $\square$

Proposition 4.1 Given a database instance D, and a set IC of ICs, $\Pi'(D, IC)$ has the same stable models as $\Pi(D, IC)$.

Proof: From Definition 4.1 we know that $SM(\Pi(D, IC)) \subseteqq SM(\Pi'(D, IC))$, hence we need to show that $SM(\Pi'(D, IC)) \subseteqq SM(\Pi(D, IC))$.

By contradiction, let us assume that there exists a stable model $\mathcal{M}'$ for program $\Pi'(D,\ IC)$ that is not a model of $\Pi(D, IC)$. $\mathcal{M}'$ is an incoherent model, otherwise it would be a model of $\Pi(D, IC)$. Therefore, for a given predicate P, $\mathcal{M}'$ contains both

$P\_(\bar{a}, \mathbf{t_a})$, and $P\_(\bar{a}, \mathbf{f_a})$, and there is no a program constraint of the form: $\leftarrow P\_(\bar{a}, \mathbf{t_a})$, $P\_(\bar{a}, \mathbf{f_a})$ in program $\Pi'(D, IC)$. Since there is no a program constraint for predicate P in program $\Pi'(D, IC)$, either it is a sink or a source node in the dependency graph. Nevertheless, if it is a sink (source) node, then there is no a rule in program $\Pi'(D, IC)$ having $P\_(\bar{a}, \mathbf{f_a})$ in its head ($P\_(\bar{a}, \mathbf{t_a})$ if source), therefore $P\_(\bar{a}, \mathbf{f_a})$ ($P\_(\bar{a}, \mathbf{t_a})$ if source) cannot be true in model $\mathcal{M}'$. But $P\_(\bar{a}, \mathbf{t_a})$ is in $\mathcal{M}'$. We have reached a contradiction.
$\square$

Moreover, it is possible to identify classes of ICs for which repair programs do not have any program constraints.

Corollary 4.1 If set IC of ICs contains only formulas of the form $\bar{\forall}(\bigwedge_{i=1}^{n} P_i\ (\bar{x}_i) \rightarrow \varphi)$, where $P_i(\bar{x}_i)$ is an atom and φ is a formula containing built-ins only, or NNCs of the form (2.6), then the dependence graph $\mathcal{G}(IC)$ has only sink nodes. In consequence, the repair program $\Pi'(D, IC)$ has no program constraints.

Proof: Directly from the dependence graph $\mathcal{G}(IC)$. $\square$

This corollary includes important classes of ICs such as key constraints, functional dependencies, unary constraints, i.e. constraints of the form (2.2), with one database predicate in the antecedent of the IC and only a built-in formula in the consequent, and also NNCs. For all these ICs, we can evaluate programs with magic sets options directly in the DLV system. In Chapter 5 we show that the magic sets technique improves considerably the bottom-up evaluation of programs.

4.4 Optimized Repair Programs

By using all the transformations introduced so far, we obtain a new definition for the repair program.

Definition 4.2 Given a database instance D, a set IC of UICs, RICs, and NNCs, the repair program $\Pi^\star(D,\, IC)$ contains:

1. $P(\bar{a})$, for each atom $P(\bar{a}) \in D$.

2. For every UIC ψ of the form (2.3), the set of rules:

   $\bigvee_{i=1}^{n} P\_i(\bar{x}_i, \mathbf{f_a}) \vee \bigvee_{j=1}^{m} Q\_j(\bar{y}_j, \mathbf{t_a}) \;\leftarrow\; \bigwedge_{i=1}^{n} P\_i(\bar{x}_i, \mathbf{t}^\star), \bigwedge_{Q\_j \in Q'} Q\_j(\bar{y}_j, \mathbf{f_a}),$

   $\bigwedge_{Q_k \in Q''}$ *not* $Q_k(\bar{y}_k), \bigwedge_{x_l \in \mathcal{A}(\psi) \cap \bar{x}} x_l \neq \mathit{null}, \bar{\varphi},$

   for every set Q' and Q'' of atoms appearing in formula (2.3) such that $Q' \cup Q'' = \bigcup_{j=1}^{m} Q_j(\bar{y_j})$ and $Q' \cap Q'' = \emptyset$, where $\mathcal{A}(\psi)$ is the set of relevant attributes for ψ, $\bar{x} = \bigcup_{i=1}^{n} x_i$, and $\bar{\varphi}$ is a conjunction of built-ins that is equivalent to the negation of φ.

3. For every RIC ψ of the form (2.4), the rules:

   $P\_(\bar{x}, \mathbf{f_a}) \vee Q\_(\bar{y}, \overline{null}, \mathbf{t_a}) \leftarrow P\_(\bar{x}, \mathbf{t}^\star),$ *not* $aux_\psi(\bar{y}),\ \bar{y} \neq \mathit{null}.$

   And for every $z_i \in \bar{z}$:

   $aux_\psi(\bar{y}) \leftarrow Q\_(\bar{y}, \bar{z}, \mathbf{t}^\star),$ *not* $Q\_(\bar{y}, \bar{z}, \mathbf{f_a}),\ \bar{y} \neq \mathit{null},\ z_i \neq \mathit{null}.$

4. For every NNC of the form (2.6), the rule:

   $P\_(\bar{x}, \mathbf{f_a}) \leftarrow P\_(\bar{x}, \mathbf{t}^\star), x_i = \mathit{null}.$

5. For each predicate $P \in R$, the annotation rules:

   $P\_(\bar{x}, \mathbf{t}^\star) \leftarrow P(\bar{x}). \quad P\_(\bar{x}, \mathbf{t}^\star) \leftarrow P\_(\bar{x}, \mathbf{t_a}).$

6. For every predicate $P \in \mathcal{R}$, the interpretation rule:

   $P\_(\bar{x}, \mathbf{t}^{\star\star}) \;\leftarrow\; P\_(\bar{x}, \mathbf{t}^\star),$ *not* $P\_(\bar{x}, \mathbf{f_a}).$

7. For every predicate $P \in \mathcal{R}$ that is not a sink or a source node in $\mathcal{G}(IC)$, the program constraint: $\leftarrow\ P\_(\bar{x}, \mathbf{t_a}),\ P\_(\bar{x}, \mathbf{f_a}).$ □

As we would expect, the new optimized repair programs produce the same database repairs than the original ones.

Theorem 4.1 Given a database instance D, and a set IC of UICs, RICs, and NNCs of the forms (2.3), (2.4) and (2.6), respectively, the repair program $\Pi(D, IC)$ as in Definition 2.10 and the program $\Pi^\star(D, IC)$ produce the same database repairs. $\Box$

Before proving the theorem, we introduce some concepts and obtain some results that are needed for the proof of Theorem 4.1.

Definition 4.3 [29] Given a model M of a program Π, a predicate symbol P, and a set of interpretations I:

(a) $M[P]$ denotes the set of atoms in M whose predicate symbol is P.

(b) $\Pi[P]$ is the set of rules of Π whose head contains predicate P.

(c) $M[\Pi]$ is the set of atoms in M whose predicate symbol appears in the head of some rule in program Π.

(d) $I[P] = \{M[P] \mid M \in I\}$, $I[\Pi] = \{M[\Pi] \mid M \in I\}$. $\Box$

Intuitively, we first prove (cf. Proposition 4.2) that the elimination of annotation constants and rules from repair programs does not affect the semantics of programs. Second, we show that the interpretation rules of programs $\Pi(D, IC)$ and $\Pi^\star(D, IC)$ generate the same atoms annotated with $\mathbf{t}^{\star\star}$, which are the atoms that become true in the repairs.

For the proof of Proposition 4.2, and Theorem 4.1 we simplify the notation as follows: Π denotes the repair program $\Pi(D, IC)$, and $\Pi^\star$ denotes the repair program $\Pi^\star(D, IC)$. Also, we define two sets: AC contains the following annotation constants $\mathbf{f_a}$, $\mathbf{t_a}$, $\mathbf{t_d}$, $\mathbf{t}^\star, \{\}$, where $\{\}$ is used to indicate "no annotation" of database facts in program $\Pi^\star$. The second set is IR which only contains constant $\mathbf{t}^{\star\star}$. $SM(\mathit{\Pi})[AC]$ is the set of stable models of program Π restricted to the atoms that have annotation constants in AC (for $\Pi^\star$, $SM(\Pi^\star)[AC]$ contains also database facts without annotations). $\Pi[AC]$ is the repair program restricted to the rules whose head atom contains one of the annotation arguments in AC. $\Pi[IR]$ denotes the interpretation rules of the repair program. Consequently, $\Pi = \Pi[IR] \cup \Pi[AC]$.

Proposition 4.2 M is a stable model of $\Pi[AC]$ iff M' is a stable model of $\Pi^\star[AC]$ with $M = M'$ wrt the atoms with annotation constants $\mathbf{f_a}$, $\mathbf{t_a}$, $\mathbf{t^\star}$.

Proof: We just need to prove that the elimination of annotation $\mathbf{t_d}$, *dom* atoms, and program constraints does not affect the semantics of repair programs and, therefore programs Π and Π^* have the same stable models, restricted to atoms with annotation constants $\mathbf{f_a}$, $\mathbf{t_a}$, $\mathbf{t^\star}$.

(a) The elimination of annotation $\mathbf{t_d}$ of repair programs does not affect the semantics of the program.

In program $\Pi^\star$ annotation $\mathbf{t_d}$ is eliminated and database facts are used as they come from the database, e.g. they are atoms of the form $P(\bar{a})$. It is easy to see that $P(\bar{a}, \mathbf{t_d})$ and $P(\bar{a})$ refer to the same database facts, since they are retrieved from the same database instance D. Moreover, due to the elimination of $\mathbf{t_d}$ annotation, in program $\Pi^\star$ the version of P that is expanded with other annotations is replaced by an underscored version, e.g. $P(\bar{a}, \mathbf{t_a})$, becomes $P\_(\bar{a}, \mathbf{t_a})$, etc. This is just a syntactic change. Therefore, the elimination of annotation $\mathbf{t_d}$ does not alter the semantics of the repair program.

(b) The replacement of $dom(\bar{x})$ in rules, by conditions of the form $\bar{x} \neq null$ does not affect the semantics of the program.

dom atoms are used to capture the active domain of the database without *null*. By replacing it by $\bar{x} \neq null$, we ensure that x does not take the null value. Also, since stable models are minimal models, they only consider the constants in the active domain. As a consequence, we can eliminate *dom* of repair programs without affecting their semantics.

(c) The elimination of program constraints as established in Proposition 4.1 does not affect the semantics of the program. $\square$

Proof of Theorem 4.1: Having Proposition 4.2, we just need to prove that the interpretation rules of programs $\Pi^\star$ and Π define the same atoms annotated with $\mathbf{t}^{\star\star}$.

It is easy to see that the programs Π and $\Pi^\star$ can be split [63] into a bottom program $\Pi[AC]$ (resp. $\Pi^\star[AC]$) and a top program $\Pi[IR]$ (resp. $\Pi^\star[IR]$), using as a splitting set all the atoms except the ones annotated with $\mathbf{t}^{\star\star}$. This implies that the programs can be hierarchically evaluated in the following way: The models of program Π are $SM(\Pi) = \bigcup_M SM(M \cup \Pi[IR])$, for each stable model M in $SM(\Pi)[AC]$.

Therefore, now we will prove that:

(1) For every stable model M" that belongs to $SM(M \cup \Pi[IR])$ with M in $SM(\Pi)[AC]$, there exists a stable model $M^\star$ that belongs to $SM(M' \cup \Pi^\star[IR])$, with M' in $SM(\Pi^\star)[AC]$, such that M"$[\mathbf{t}^{\star\star}] = M^\star[\mathbf{t}^{\star\star}]$, that is, M" and $M^\star$ contain the same atoms annotated with $\mathbf{t}^{\star\star}$.

By contradiction, let us assume that there exists a stable model M" in $SM(M \cup \Pi[IR])$ with M in $SM(\Pi)[AC]$, and there is not a stable model $M^\star$ that belongs to $SM(M' \cup \Pi^\star[IR])$ with M' in $SM(\Pi^\star)[AC]$, such that M"$[\mathbf{t}^{\star\star}] = M^\star[\mathbf{t}^{\star\star}]$.

We have two cases depending if the repair obtained from M" is empty or not.

(a) The repair obtained from M" is empty. So M" does not have atoms with the $\mathbf{t}^{\star\star}$ constant. In this case M" $= M$ with M in $SM(\Pi)$ $[AC]$. Then, according to Proposition 4.2 we know that there exists a model M' in $SM(\Pi^\star)$ $[AC]$ such that $M = M'$. Therefore, given the fact that M" $= M$, we now also have that

M" $= M'$. Now, if $M^\star$ has no atoms with $\mathbf{t^{\star\star}}$, M"$[\mathbf{t^{\star\star}}]$ would be equal to $M^\star[\mathbf{t^{\star\star}}]$ (both would be empty) and this would lead to a contradiction. Then, $M^\star[\mathbf{t^{\star\star}}]$ should not be empty. Hence, there exists an atom $P\_(\bar{c}, \mathbf{t^{\star\star}})$ in $M^\star$. Then, M' has $P\_(\bar{c}, \mathbf{t^\star})$ and does not have $P\_(\bar{c}, \mathbf{f_a})$. If M' has $P\_(\bar{c}, \mathbf{t^\star})$ then $P(\bar{c})$ is true or $P\_(\bar{c}, \mathbf{t_a})$ is true in M'. However, if either of both situations happens, and given the facts that $P\_(\bar{c}, \mathbf{f_a})$ is false in M', and $M = M'$, then M" satisfies $P(\bar{c}, \mathbf{t^{\star\star}})$ as well. Because of the interpretation rules in $\Pi[IR]$. But, M"$[\mathbf{t^{\star\star}}] = \emptyset$. We have reached a contradiction.

(b) The repair obtained from M" is not empty. In this case, there exists an atom $P(\bar{c}, \mathbf{t^{\star\star}})$ in M" such that there is no model $M^\star$ that satisfies $P\_(\bar{c}, \mathbf{t^{\star\star}})$. If atom $P(\bar{c}, \mathbf{t^{\star\star}})$ is true in M" then we have that $P(\bar{c}, \mathbf{t_d})$ is in M, (M in $SM(\Pi)[AC]$) in which case $P(\bar{c}, \mathbf{f_a})$ is not in M, or $P(\bar{c}, \mathbf{t_d})$ is not in M, in which case $P(\bar{c}, \mathbf{t_a})$ is in M. In both cases we have that atom $P(\bar{c}, \mathbf{t^\star})$ is true in M. In addition, because of Proposition 4.2 we know that exists a stable model M' in $SM(\Pi^\star)[AC]$, such that $M = M'$. Now, there are two cases to consider: $P(\bar{c}, \mathbf{t_d})$ is in M and $P(\bar{c}, \mathbf{t_d})$ is not in M.

First, for $P(\bar{c}, \mathbf{t_d})$ in M, we have that since $P(\bar{c}, \mathbf{t^\star})$ is in M, $P(\bar{c}, \mathbf{f_a})$ has to be false in M. Then, since $M = M'$ and because of the interpretation rule in $\Pi^\star$ $P\_(\bar{c}, \mathbf{t^{\star\star}}) \leftarrow P\_(\bar{c}, \mathbf{t^\star})$, *not* $P\_(\bar{c}, \mathbf{f_a})$, we have that there exists a model $M^\star$ such that $P\_(\bar{c}, \mathbf{t^{\star\star}})$ is in $M^\star$. Then we have reached a contradiction. Second, for $P(\bar{c}, \mathbf{t_d})$ not in M, we have that since $P(\bar{c}, \mathbf{t^\star})$ is in M, $P(\bar{c}, \mathbf{t_a})$ has to be true in M and $P(\bar{c}, \mathbf{f_a})$ has to be false in M. Then, since $M = M'$ and because of the interpretation rule in $\Pi^\star$ $P\_(\bar{c}, \mathbf{t^{\star\star}}) \leftarrow P\_(\bar{c}, \mathbf{t^\star})$, *not* $P\_(\bar{c}, \mathbf{f_a})$, we have that there exists a model $M^\star$ such that $P\_(\bar{c}, \mathbf{t^{\star\star}})$ is in $M^\star$. We have reached a contradiction.

(2) For every stable model $M^\star$ that belongs to $\mathit{SM}(M' \cup \Pi^\star[\mathit{IR}])$, with M' in $\mathit{SM}(\Pi^\star)[AC]$, there exists a stable model M" that belongs to $\mathit{SM}(M \cup (\Pi)[\mathit{IR}])$ with M in $\mathit{SM}(\Pi)[AC]$, such that $M^\star[\mathbf{t^{\star\star}}] = M$"$[\mathbf{t^{\star\star}}]$.

By contradiction, let us assume that there exists a model $M^\star$ that belongs to $\mathit{SM}(M' \cup \Pi^\star[\mathit{IR}])$ with M' in $\mathit{SM}(\Pi^\star)[AC]$, and there is not a stable model M" that belongs to $\mathit{SM}(M \cup \Pi[\mathit{IR}])$ with M in $\mathit{SM}(\Pi)[AC]$, such that $M^\star[\mathbf{t^{\star\star}}] = M$"$[\mathbf{t^{\star\star}}]$.

Here we have two cases depending if the repair obtained from $M^\star$ is empty or not.

(a) The repair obtained from $M^\star$ is empty. So $M^\star$ does not have atoms with the $\mathbf{t^{\star\star}}$ constant. In this case $M^\star = M'$ with M' in $\mathit{SM}(\Pi^\star)[AC]$. Then, because of Proposition 4.2 we know that there exists a model M in $\mathit{SM}(\Pi)$ $[AC]$ such that $M' = M$. So given the fact that $M^\star = M'$, we now also have that $M^\star = M$. Now, if $M^\star$ has no atoms with $\mathbf{t^{\star\star}}$, $M^\star[\mathbf{t^{\star\star}}]$ would be equal to M"$[\mathbf{t^{\star\star}}]$ (both would be empty) and this would lead to a contradiction. Then, M"$[\mathbf{t^{\star\star}}]$ should not be empty.

Then, there exists an atom $P(\bar{c}, \mathbf{t^{\star\star}})$ in M". There are two cases to analyze, $P(\bar{c}, \mathbf{t_d})$ is in M or $P(\bar{c}, \mathbf{t_d})$ is not in M. First, if $P(\bar{c}, \mathbf{t_d})$ is in M, then, $P(\bar{c})$ is in M' and $P\_(\bar{c}, \mathbf{t^\star})$ is in M'. Moreover, since $P(\bar{c}, \mathbf{t^{\star\star}})$ is in M", $P(\bar{c}, \mathbf{f_a})$ is not in M and $P\_(\bar{c}, \mathbf{f_a})$ is not in M'. Given the interpretation rule in $\Pi^\star$ $P\_(\bar{c}, \mathbf{t^{\star\star}}) \leftarrow P\_(\bar{c}, \mathbf{t^\star}),\ \mathit{not}\ P\_(\bar{c}, \mathbf{f_a})$, we have $P\_(\bar{c}, \mathbf{t^{\star\star}})$ is in $M^\star$. But, $M^\star[\mathbf{t^{\star\star}}]$ is empty. We have reached a contradiction. Now, we need to analyze for $P(\bar{c}, \mathbf{t_d})$ is not in M. Since $P(\bar{c}, \mathbf{t^{\star\star}})$ is in M", $P(\bar{c}, \mathbf{f_a})$ is not in M and $P(\bar{c}, \mathbf{t_a})$ is in M. Then, given the fact that $M = M'$, then $P\_(\bar{c}, \mathbf{t_a})$ is in M', and $P\_(\bar{c}, \mathbf{t^\star})$ is in M'. Hence given the interpretation rule in $\Pi^\star$ $P\_(\bar{c}, \mathbf{t^{\star\star}}) \leftarrow P\_(\bar{c}, \mathbf{t^\star}),\ \mathit{not}\ P\_(\bar{c}, \mathbf{f_a})$, we have $P\_(\bar{c}, \mathbf{t^{\star\star}})$ is in $M^\star$. But, $M^\star[\mathbf{t^{\star\star}}]$ is empty. We have reached a contradiction.

(b) The repair obtained from $M^\star$ is not empty, so there exists $P\_(\bar{c}, \mathbf{t^{\star\star}})$ in $M^\star$. If atom $P\_(\bar{c}, \mathbf{t^{\star\star}})$ is true in $M^\star$ then we have that $P\_(\bar{c}, \mathbf{t^\star})$ is true in M' (M' is in $SM(\Pi^\star)[AC]$), and $P\_(\bar{c}, \mathbf{f_a})$ is false in M'. If $P\_(\bar{c}, \mathbf{t^\star})$ is true, then either $P(\bar{c})$ or $P\_(\bar{c}, \mathbf{t_a})$ are true in M'. In addition, because of Proposition 4.2 we know that exists a model M in $SM(\Pi)[AC]$, such that $M' = M$. Now there are two cases to consider: $P(\bar{c})$ is in M' or $P(\bar{c})$ is not in M'. First we will assume that $P(\bar{c})$ is true in M', and therefore that $P\_(\bar{c}, \mathbf{f_a})$ is false. Then because $M = M'$, and by using the interpretation rule $P(\bar{c}, \mathbf{t^{\star\star}}) \leftarrow P(\bar{c}, \mathbf{t_d}),\ \mathit{not}\ P(\bar{c}, \mathbf{f_a})$ of Π, $P(\bar{c}, \mathbf{t^{\star\star}})$ is in M". Then M" exists and we have reached a contradiction. Now, if $P(\bar{c})$ is not in M' we have that $P\_(\bar{c}, \mathbf{t_a})$ is in M'. Then because $M = M'$, and by using the interpretation rule $P(\bar{c}, \mathbf{t^{\star\star}}) \leftarrow P(\bar{c}, \mathbf{t_a})$ of Π, $P(\bar{c}, \mathbf{t^{\star\star}})$ is in M". Then M" exists and we have reached a contradiction. □

Example 4.4 The repair program $\Pi^\star(D, IC)$ for $D = \{P(a, b, \mathit{null}), P(b, b, a), R(a, b), S(a, b, b), S(\mathit{null}, b, b)\}$, and IC: $\forall xyz(P(x, y, z) \rightarrow R(x, y))$, $\forall xy(R(x, y) \rightarrow \exists z S(x, y, z))$, and $\forall xyz(S(x, y, z) \wedge \mathit{IsNull}(x) \rightarrow \mathbf{false})$ contains the following facts and rules:

1. $P(a, b, \mathit{null}).\quad P(b, b, a).\quad R(a, b).\quad S(a, b, b).\quad S(\mathit{null}, b, b).$
2. $P\_(x, y, z, \mathbf{f_a}) \vee R\_(x, y, \mathbf{t_a}) \leftarrow P\_(x, y, z, \mathbf{t^\star}), R\_(x, y, \mathbf{f_a}), x \neq \mathit{null}, y \neq \mathit{null}.$

   $P\_(x, y, z, \mathbf{f_a}) \vee R\_(x, y, \mathbf{t_a}) \leftarrow P\_(x, y, z, \mathbf{t^\star}),\ \mathit{not}\ R(x, y), x \neq \mathit{null}, y \neq \mathit{null}.$
3. $R\_(x, y, \mathbf{f_a}) \vee S\_(x, y, \mathit{null}, \mathbf{t_a}) \leftarrow R\_(x, y, \mathbf{t^\star}),\ \mathit{not}\ \mathit{aux}(x, y), x \neq \mathit{null}, y \neq \mathit{null}.$

   $\mathit{aux}(x, y) \leftarrow S\_(x, y, z, \mathbf{t^\star}),\ \mathit{not}\ S\_(x, y, z, \mathbf{f_a}), x \neq \mathit{null}, y \neq \mathit{null}, z \neq \mathit{null}.$
4. $S\_(x, y, z, \mathbf{f_a}) \leftarrow S\_(x, y, z, \mathbf{t^\star}), x = \mathit{null}.$
5. $P\_(x, y, z, \mathbf{t^\star}) \leftarrow P(x, y, z).$

   $P\_(x, y, z, \mathbf{t^\star}) \leftarrow P\_(x, y, z, \mathbf{t_a}).$

   $R\_(x, y, \mathbf{t^\star}) \leftarrow R(x, y).$

   $R\_(x, y, \mathbf{t^\star}) \leftarrow R\_(x, y, \mathbf{t_a}).$

   $S\_(x, y, z, \mathbf{t^\star}) \leftarrow S(x, y, z).$

   $S\_(x, y, z, \mathbf{t^\star}) \leftarrow S\_(x, y, z, \mathbf{t_a}).$

6. $P\_(x,y,z,\mathbf{t^{\star\star}}) \leftarrow P\_(x,y,z,\mathbf{t^\star}),\ \textit{not}\ P\_(x,y,z,\mathbf{f_a})$.

$R\_(x,y,\mathbf{t^{\star\star}}) \leftarrow R\_(x,y,\mathbf{t^\star}),\ \textit{not}\ R\_(x,y,\mathbf{f_a})$.

$S\_(x,y,z,\mathbf{t^{\star\star}}) \leftarrow S\_(x,y,z,\mathbf{t^\star}),\ \textit{not}\ S\_(x,y,z,\mathbf{f_a})$.

7. $\leftarrow S\_(x,y,z,\mathbf{t_a}), S\_(x,y,z,\mathbf{f_a})$.

Note that program constraints are only generated for predicate S since for this predicate there are rules generating atoms with both $\mathbf{t_a}$ and $\mathbf{f_a}$ annotations. This program has two stable models:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{P(a,b,\textit{null}), P(b,b,a), R(a,b), S(a,b,b), S(\textit{null},b,b), P\_(a,b,\textit{null},\mathbf{t^\star}),\\
& P(b,b,a,\mathbf{t^\star}), R\_(a,b,\mathbf{t^\star}), S\_(a,b,b,\mathbf{t^\star}), S\_(\textit{null},b,b,\mathbf{f_a}), S\_(\textit{null},b,b,\mathbf{t^\star}),\\
& \textit{aux}(a,b), R\_(b,b,\mathbf{t_a}), \underline{P\_(a,b,\textit{null},\mathbf{t^{\star\star}})}, \underline{P\_(b,b,a,\mathbf{t^{\star\star}})}, \underline{R\_(a,b,\mathbf{t^{\star\star}})}, R\_(b,b,\mathbf{t^\star}),\\
& S\_(b,b,\textit{null},\mathbf{t_a}), \underline{R\_(b,b,\mathbf{t^{\star\star}})}, S\_(b,b,\textit{null},\mathbf{t^\star}), \underline{S\_(b,b,\textit{null},\mathbf{t^{\star\star}})}, \underline{S\_(a,b,b,\mathbf{t^{\star\star}})}\},\\
\mathcal{M}_2 \;=\; & \{P(a,b,\textit{null}), P(b,b,a), R(a,b), S(a,b,b), S(\textit{null},b,b), P\_(a,b,\textit{null},\mathbf{t^\star}),\\
& P\_(b,b,a,\mathbf{t^\star}), R\_(a,b,\mathbf{t^\star}), S\_(a,b,b,\mathbf{t^\star}), S\_(\textit{null},b,b,\mathbf{f_a}), S\_(\textit{null},b,b,\mathbf{t^\star}),\\
& \textit{aux}(a,b), P\_(b,b,a,\mathbf{f_a}), \underline{P\_(a,b,\textit{null},\mathbf{t^{\star\star}})}, \underline{R\_(a,b,\mathbf{t^{\star\star}})}, \underline{S\_(a,b,b,\mathbf{t^{\star\star}})}\}.
\end{aligned}$$

Stable models of program $\Pi^\star(D, IC)$ contain less predicates than the models of $\Pi(D, IC)$ (cf. Example 2.9), which is due to the fact that *dom* predicate was eliminated from repair programs. Moreover, they construct the same database repairs, as expected. □

From now on, repair programs are those given in Definition 4.2, and they will be denoted just by $\Pi(D, IC)$, as before.

Moreover, for a database D and a RIC-acyclic set IC of UICs, RICs, and NNCs, the program $\Pi(D, IC)$ without its program constraints is *locally stratified.*

Definition 4.4 [68] A program Π is *locally stratified* if its Herbrand base can be partitioned into sets $S_0, S_1, \ldots$ (called strata) such that, for each rule

$$A_1 \vee \cdots \vee A_l \leftarrow B_1, \ldots, B_m,\ not\ C_1, \ldots,\ not\ C_n,$$

in $ground(\Pi)$, there exists an $i \geq 1$ such that all $A_1, \ldots, A_l$ belong to S_i, all $B_1, \ldots, B_m$, belong to $S_0 \cup \cdots \cup S_i$, and all $C_1, \ldots, C_n$ belong to $S_0 \cup \cdots \cup S_{i-1}$. For such a partition, we use Π_i to denote the set of all rules from $ground(\Pi)$ whose consequent belong to S_i. □

The notion of *locally stratified* programs differs from the definition of *stratified* programs.

Definition 4.5 [66] A program Π is *stratified* if there exists a level mapping $\| \quad \|$ from predicates (names) of Π to natural numbers, such that for every rule r of Π:

(a) For any positive literal $l \in B^+(r)$, and for any literal l' in $H(r)$, $\| l \| \leq \| l' \|$

(b) For any negative literal $l \in B^-(r)$, and for any literal l' in $H(r)$, $\| l \| < \| l' \|$□

The difference between *locally stratified* and *stratified* programs is that in the former the strata are generated by considering atoms on the ground version of program Π. However, in the latter the strata are generated by considering the set of predicates appearing in program Π.

We will use the next result in Chapter 5, to prove that we can use the magic sets technique in the evaluation of repair programs. The magic set method we define is applied to repair programs without considering its program constraints.

Proposition 4.3 For a database instance D and a RIC-acyclic set IC of UICs, RICs and NNCs of the forms (2.3), (2.4), and (2.6), respectively, the repair program $\Pi(D, IC)$ without its program constraints is locally stratified.

Proof: Given a database D and a set of RIC-acyclic UICs, RICs, and NNCs, let $\{V_1, \ldots, V_r\}$ be the set of vertices of $\mathcal{G}^C(IC)$. Since this graph is obtained by contracting vertices of $\mathcal{G}(IC)$, each vertex in $\mathcal{G}^C(IC)$ is a set of predicates of $\mathcal{R}$. In fact, $\bigcup_{i=1}^{r} V_i = \mathcal{R}$, and $V_j \cap V_k = \emptyset$, for $V_j, V_k \in V$. Since $\mathcal{G}^C(IC)$ is acyclic, we can safely assume that the vertices are numbered in a topological ordering, i.e. for every edge (V_i, V_j), we have $i < j$. Then, for $\Pi'(D, IC) := \Pi(D, IC) \smallsetminus PC$, where PC is the set of program constraints in program $\Pi(D, IC)$, we can consider the following *strata*:

$S_0 = \{P(\bar{x}) \mid P \in \mathcal{R} \text{ and } \bar{x} \in \mathcal{U}\}$,

$S_1 = \{P\_(\bar{x}, y) \mid P \in \mathcal{V}_r, \bar{x} \in \mathcal{U}, \text{ and } y \in \{\mathbf{t}^\star, \mathbf{t_a}, \mathbf{f_a}\}\}$,

$S_2 = \{aux_i(\bar{x}) \mid V_r$ has an incoming edge in $\mathcal{G}^C(IC)$ corresponding to the referential integrity constraint IC_i, and $\bar{x} \in \mathcal{U}\}$,

$S_3 = \{P\_(\bar{x}, y) \mid P \in \mathcal{V}_{r-1}, \bar{x} \in \mathcal{U}, \text{ and } y \in \{\mathbf{t}^\star, \mathbf{t_a}, \mathbf{f_a}\}\}$,

$S_4 = \{aux_i(\bar{x}) \mid V_r$ has an incoming edge in $\mathcal{G}^C(IC)$ corresponding to the referential integrity constraint IC_i, and $\bar{x} \in \mathcal{U}\}$,

...

$S_i = \{P\_(\bar{x}, y) \mid P \in \mathcal{V}_{r-\lfloor\frac{i-1}{2}\rfloor}, \bar{x} \in \mathcal{U}, \text{ and } y \in \{\mathbf{t}^\star, \mathbf{t_a}, \mathbf{f_a}\}\}$, for $i \leq (2r-1)$ and odd,

$S_i = \{aux_i(\bar{x}) \mid V_{r-\lfloor\frac{i-1}{2}\rfloor}$ has an incoming edge in $\mathcal{G}^C(IC)$ corresponding to the RIC IC_i, and $\bar{x} \in \mathcal{U}\}$, for $i \leq (2r-1)$ and even,

...

$S_{2r-1} = \{P\_(\bar{x}, y) \mid P \in \mathcal{V}_1, \bar{x} \in \mathcal{U}, \text{ and } y \in \{\mathbf{t}^\star, \mathbf{t_a}, \mathbf{f_a}\}\}$,

$S_{2r} = \{P\_(\bar{x}, y) \mid P \in \mathcal{R}, \bar{x} \in \mathcal{U}, \text{ and } y \in \{\mathbf{t}^{\star\star}\}\}$

It is easy to check that this stratification satisfies the conditions on every rule of program $\Pi'(D, IC)$ and, therefore program $\Pi'(D, IC)$ is locally stratified. □

Notice that, since there is no recursion in FO queries, the query program $\Pi(\mathcal{Q})$ is always stratified, then it holds that program $\Pi(D, IC) \cup \Pi(\mathcal{Q})$ is locally stratified.

Proposition 4.4 For a database instance D, a RIC-acyclic set IC of UICs, RICs and NNCs of the forms (2.3), (2.4), and (2.6), respectively, and a FO query $\mathcal{Q}$, program $\Pi(D, IC, \mathcal{Q}) := \Pi(D, IC) \cup \Pi(\mathcal{Q})$ without its program constraints is locally stratified.

Proof: Consider the strata for program $\Pi'(D, IC) := \Pi(D, IC) \smallsetminus PC$, i.e. the repair program without its program constraints, defined in the proof of Proposition 4.3. Since program $\Pi(\mathcal{Q})$ only has rules with the head atom Ans, which is not present anywhere else in program $\Pi'(D, IC)$ (and neither in $\Pi(D, IC)$), and atoms of the form $P\_(\bar{x}, \mathbf{t}^{\star\star})$ and *not* $P\_(\bar{x}, \mathbf{t}^{\star\star})$ in their bodies. The program $\Pi'(D, IC, \mathcal{Q}) := \Pi'(D, IC) \cup \Pi(\mathcal{Q})$ can be evaluated by considering the additional *stratum* $S_{2r+1} = \{Ans(\bar{x}) \mid \bar{x} \in \mathcal{U}\}$. Therefore, program $\Pi'(D, IC, \mathcal{Q})$ is locally stratified. $\Box$

4.5 Summary

In this chapter repair programs have been simplified and optimized by eliminating redundant rules, facts and annotations. It is important to remark that the processing of database facts has been eliminated. Now database facts are used as they come from the database, without adding any annotation. In Chapter 9 we will see that the elimination of the $\mathbf{t_d}$ constant permits to import database facts directly into the reasoning system.

Moreover, it was shown that program constraints, which are the rules that avoid the generation of incoherent models, are not always needed. Thus, we defined a methodology to detect database predicates for which program constraints should not be generated. Their elimination is important because it avoids unnecessary model checking in a reasoning system. In addition, important classes of ICs are identified for which repair programs can be specified without program constraints. It becomes relevant when magic sets techniques are applied with the DLV system (cf. Chapter 5), which implements magic sets for programs without program constraints.

Furthermore, it was proven that the new optimized programs (without program constraints) are locally stratified. This result will be used in Chapter 5 to prove that the MS methodology we define is sound and complete.

Chapter 5

Optimizing Query Evaluation from Repair Programs

5.1 Introduction

In this chapter we present optimizations of the evaluation of repair and query programs to obtain consistent answers.

Consistent answers are obtained from stable models of the combination of the repair and query programs. Nevertheless, in most of the cases the former -so as its stable models- contain more information than necessary to answer the query, because repair programs are built considering all database predicates and database facts. However, query predicates are related to a subset of the database predicates.

Furthermore, we are not interested in obtaining complete stable models (or repairs), but only in obtaining the consistent answers to our queries. In consequence, it is important to optimize the evaluation of the repair programs by considering only predicates and facts that are relevant to the query. This is precisely the purpose of the magic sets (MS) technique [8], that achieves it by simulating a top-down [23] -and then query directed- evaluation of the query through bottom-up propagation [23]. This technique produces a new program that contains a subset of the original rules, along with a set of new, "magic", rules.

Classic MS techniques for Datalog programs [8, 70] have been extended to logic programs with unstratified negation under stable models semantics [35], to disjunctive logic programs with stratified negation [45], with an optimized version [29] being implemented in DLV. For this kind of programs, the MS technique is sound and complete, i.e. the method computes all and only correct answers for the query. We know by personal communication with the authors of [29, 35], that the same result can be obtained as a combination of [29, Theorem 1] and [35, Theorem 3]. This result holds for disjunctive programs with possibly unstratified negation, under certain conditions (cf. Section 5.2). In [47] a sound but incomplete methodology is presented for disjunctive programs with constraints of the form $\leftarrow C(\bar{x})$, where $C(\bar{x})$ is a conjunction of literals (i.e. positive or negated atoms).

In this chapter, we present a sound and complete MS methodology for our disjunctive repair programs with their program constraints, which fall in the category of program constraints with only positive intensional literals in the body. The methodology works for the kind of programs we have, but not necessarily in the general case of disjunctive programs with program constraints.

It works as follows: the set of program constraints PC is separated from the rest of the rules, then the MS technique, as defined in [29, 35], is applied to the resulting program. The latter is possible, since our disjunctive programs with program constraints satisfy the conditions to apply MS as defined in [29, 35] (we explain this in detail in Section 5.2). At the end of this process, the program constraints are put back into the resulting program, and so enforcing that the rewritten program has only coherent models.

Moreover, we develop another optimization technique that can be used as an alternative to the MS method. This technique also captures the relevant database predicates to compute a specific query. The relevant predicates are used to generate reduced repair programs, i.e. programs that consider the relevant predicates to compute a query. This new repair program is evaluated together with the query program. It is shown that the relevant predicates produced by this method correspond to the predicates selected by the MS technique. In Chapter 9 we perform experiments to show the effectiveness of the relevant predicates methodology for CQA, and we compare it with the MS technique for query evaluation.

Optimizations to the evaluation of programs make query processing more efficient. In general, consistent query answering over inconsistent databases is an expensive computational task (worst case Π_2^P-complete in data complexity [17, 27]). Therefore, speeding up query evaluation over large data sets becomes particularly relevant.

The rest of the chapter is organized as follows: Section 5.2 introduces the magic sets methodology for disjunctive repair programs with program constraints. In Section 5.3 we specify how to apply magic sets in the DLV systems to our repair programs. In Section 5.4 we present a method to generate reduced repair programs based on relevance of predicates. In Section 5.5 we discuss related work on optimizations of the logic approach for CQA. Section 5.6 finalizes this chapter.

5.2 Magic Sets for Repair Programs

Given a query and a program, the MS selects the relevant rules from the program to compute the answers to the query, and pushes down the constants in the query to restrict the tuples involved in the computation of the answers. The MS methodology carries this out by sequentially performing three well defined steps: *adornment, generation and modification.* The method will be illustrated using the following repair program and query, where rules have been enumerated for reference.

Example 5.1 Given a database instance $D = \{S(a), T(a)\}$, a set IC with $\forall x(S(x) \rightarrow Q(x))$, $\forall x(Q(x) \rightarrow R(x))$ and $\forall x(T(x) \rightarrow W(x))$, and query $\mathcal{Q}$: $Ans(x) \leftarrow S(x)$. The program $\Pi(D, IC, \mathcal{Q}) := \Pi(D, IC) \cup \Pi(\mathcal{Q})$ consists of the rules:

1. $S(a)$. $T(a)$.
2. $S\_(x,\mathbf{f_a}) \vee Q\_(x,\mathbf{t_a}) \leftarrow S\_(x,\mathbf{t}^\star), Q\_(x,\mathbf{f_a}),\ x \neq \mathit{null}.$
3. $S\_(x,\mathbf{f_a}) \vee Q\_(x,\mathbf{t_a}) \leftarrow S\_(x,\mathbf{t}^\star),\ \mathit{not}\ Q(x),\ x \neq \mathit{null}.$
4. $Q\_(x,\mathbf{f_a}) \vee R\_(x,\mathbf{t_a}) \leftarrow Q\_(x,\mathbf{t}^\star), R\_(x,\mathbf{f_a}),\ x \neq \mathit{null}.$
5. $Q\_(x,\mathbf{f_a}) \vee R\_(x,\mathbf{t_a}) \leftarrow Q\_(x,\mathbf{t}^\star),\ \mathit{not}\ R(x),\ x \neq \mathit{null}.$
6. $T\_(x,\mathbf{f_a}) \vee W\_(x,\mathbf{t_a}) \leftarrow T\_(x,\mathbf{t}^\star), W\_(x,\mathbf{f_a}),\ x \neq \mathit{null}.$
7. $T\_(x,\mathbf{f_a}) \vee W\_(x,\mathbf{t_a}) \leftarrow T\_(x,\mathbf{t}^\star),\ \mathit{not}\ W(x),\ x \neq \mathit{null}.$
8. $S\_(x,\mathbf{t}^\star) \leftarrow S\_(x,\mathbf{t_a})$. 9. $S\_(x,\mathbf{t}^\star) \leftarrow S(x)$.
10. $Q\_(x,\mathbf{t}^\star) \leftarrow Q\_(x,\mathbf{t_a})$. 11. $Q\_(x,\mathbf{t}^\star) \leftarrow Q(x)$.
12. $R\_(x,\mathbf{t}^\star) \leftarrow R\_(x,\mathbf{t_a})$. 13. $R\_(x,\mathbf{t}^\star) \leftarrow R(x)$.
14. $T\_(x,\mathbf{t}^\star) \leftarrow T\_(x,\mathbf{t_a})$. 15. $T\_(x,\mathbf{t}^\star) \leftarrow T(x)$.
16. $W\_(x,\mathbf{t}^\star) \leftarrow W\_(x,\mathbf{t_a})$. 17. $W\_(x,\mathbf{t}^\star) \leftarrow W(x)$.
18. $S\_(x,\mathbf{t}^{\star\star}) \leftarrow S\_(x,\mathbf{t}^\star),\ \mathit{not}\ S\_(x,\mathbf{f_a})$.
19. $Q\_(x,\mathbf{t}^{\star\star}) \leftarrow Q\_(x,\mathbf{t}^\star),\ \mathit{not}\ Q\_(x,\mathbf{f_a})$.
20. $R\_(x,\mathbf{t}^{\star\star}) \leftarrow R\_(x,\mathbf{t}^\star),\ \mathit{not}\ R\_(x,\mathbf{f_a})$.
21. $T\_(x,\mathbf{t}^{\star\star}) \leftarrow T\_(x,\mathbf{t}^\star),\ \mathit{not}\ T\_(x,\mathbf{f_a})$.
22. $W\_(x,\mathbf{t}^{\star\star}) \leftarrow W\_(x,\mathbf{t}^\star),\ \mathit{not}\ W\_(x,\mathbf{f_a})$.
23. $\leftarrow Q\_(x,\mathbf{t_a}), Q\_(x,\mathbf{f_a})$.
24. $\mathit{Ans}(x) \leftarrow S\_(x,\mathbf{t}^{\star\star})$.

The stable models are:

$$\begin{aligned}
\mathcal{M}_1 &= \{T(a), S(a), T\_(a,\mathbf{t}^\star), S\_(a,\mathbf{t}^\star), Q\_(a,\mathbf{t_a}), S\_(a,\mathbf{t}^{\star\star}), Q\_(a,\mathbf{t}^\star), R\_(a,\mathbf{t_a}),\\
&\quad Q\_(a,\mathbf{t}^{\star\star}), R\_(a,\mathbf{t}^\star), R\_(a,\mathbf{t}^{\star\star}), \underline{\mathit{Ans}(a)}, W\_(a,\mathbf{t_a}), T\_(a,\mathbf{t}^{\star\star}), W\_(a,\mathbf{t}^\star),\\
&\quad W\_(a,\mathbf{t}^{\star\star})\},\\
\mathcal{M}_2 &= \{T(a), S(a), T\_(a,\mathbf{t}^\star), S\_(a,\mathbf{t}^\star), Q\_(a,\mathbf{t_a}), S\_(a,\mathbf{t}^{\star\star}), Q\_(a,\mathbf{t}^\star), R\_(a,\mathbf{t_a}),\\
&\quad Q\_(a,\mathbf{t}^{\star\star}), R\_(a,\mathbf{t}^\star), R\_(a,\mathbf{t}^{\star\star}), \underline{\mathit{Ans}(a)}, T\_(a,\mathbf{f_a})\},
\end{aligned}$$

$$\begin{array}{rcl}\mathcal{M}_3 & = & \{T(a), S(a), T\_(a, \mathbf{t}^\star), S\_(a, \mathbf{t}^\star), S\_(a, \mathbf{f_a}), W\_(a, \mathbf{t_a}), T\_(a, \mathbf{t}^{\star\star}), W\_(a, \mathbf{t}^\star),\\ & & W\_(a, \mathbf{t}^{\star\star})\},\\ \mathcal{M}_4 & = & \{T(a), S(a), T\_(a, \mathbf{t}^\star), S\_(a, \mathbf{t}^\star), S\_(a, \mathbf{f_a}), T\_(a, \mathbf{f_a})\}.\end{array}$$

Since there are no ground *Ans*-atoms in common, there are no cautious answers to the query, and then no consistent answers. □

Given the program $\Pi(D, IC, \mathcal{Q})$, the MS technique is applied to the program $\Pi^-(D, IC, \mathcal{Q}) := \Pi(D, IC, \mathcal{Q}) \smallsetminus PC$, where PC contains the program constraint: $\leftarrow Q\_(\bar{x}, \mathbf{t_a}), Q\_(\bar{x}, \mathbf{f_a})$.

For the *adornment* step, the relationship between the query predicates and the predicates of program Π^- are explicitly defined. The output of this step is a new *adorned* program, where each intensional predicate (IDB) is of the form P^A, where A is a string of letters b, f, meaning *bound* and *free*, respectively, whose length is equal to the arity of predicate P.

Starting from the given query, adornments are created. First $\Pi(\mathcal{Q}) : Ans(x) \leftarrow S\_(x, \mathbf{t}^{\star\star})$ becomes:

$$Ans^f(x) \leftarrow S\_^{fb}(x, \mathbf{t}^{\star\star}),$$

meaning that the first argument of $S\_$ is a free variable, and the second one is bound. Notice that since annotation are constants, they are always bound. The adorned predicate $S\_^{fb}$ is used to propagate bindings (adornments) onto the rules defining atoms with predicate S. For instance, $S\_^{fb}$ propagates bindings to the rules 2, 3, 8, 9 and 18. As an illustration, the non-disjunctive rules 8 and rule 9 become, respectively:

$$S\_^{fb}(x, \mathbf{t}^\star) \leftarrow S\_^{fb}(x, \mathbf{t_a}). \qquad\qquad S\_^{fb}(x, \mathbf{t}^\star) \leftarrow S(x).$$

Extensional predicates (EDB), i.e. facts as $S(x)$ in the previous rule, only bind variables and do not receive any annotation.

When an adorned predicate is in the head of a disjunctive rule, the adornments are propagated to the body literals, and to the other head atoms. For instance, the adorned predicate $S\_^{fb}$ when used in rule 2, propagates adornments over the body atoms of the rule, and to the head atom $Q\_(x, \mathbf{t_a})$. Therefore, rule 2 becomes:

$$S\_^{fb}(x, \mathbf{f_a}) \vee Q\_^{fb}(x, \mathbf{t_a}) \leftarrow S\_^{fb}(x, \mathbf{t}^\star), Q\_^{fb}(x, \mathbf{f_a}), x \neq null.$$

Note that the adorned predicate $Q\_^{fb}$ also has to be processed. Therefore, predicate $Q\_^{fb}$ produces adornments on rules defining atoms with predicate Q, i.e. in rules 2, 3, 4, 5, 10, 11, and 19. For instance, $Q\_^{fb}$ used in rule 4 produces:

$$Q\_^{fb}(x, \mathbf{f_a}) \vee R\_^{fb}(x, \mathbf{t_a}) \leftarrow Q\_^{fb}(x, \mathbf{t}^\star), R\_^{fb}(x, \mathbf{f_a}), x \neq null.$$

Again, the new adorned predicate $R\_^{fb}$ has to be processed.

After all the adornments are properly propagated, the adorned program below is generated:

Program 5.1

$Ans^f(x) \leftarrow S\_^{fb}(x, \mathbf{t}^{\star\star}).$

$S\_^{fb}(x, \mathbf{f_a}) \vee Q\_^{fb}(x, \mathbf{t_a}) \leftarrow S\_^{fb}(x, \mathbf{t}^\star),\ Q\_^{fb}(x, \mathbf{f_a}), x \neq null.$

$S\_^{fb}(x, \mathbf{f_a}) \vee Q\_^{fb}(x, \mathbf{t_a}) \leftarrow S\_^{fb}(x, \mathbf{t}^\star),\ not\ Q(x), x \neq null.$

$S\_^{fb}(x, \mathbf{t}^\star) \leftarrow S\_^{fb}(x, \mathbf{t_a}).$

$S\_^{fb}(x, \mathbf{t}^\star) \leftarrow S(x).$

$S\_^{fb}(x, \mathbf{t}^{\star\star}) \leftarrow S\_^{fb}(x, \mathbf{t}^\star),\ not\ \ S\_^{fb}(x, \mathbf{f_a}).$

$Q\_^{fb}(x, \mathbf{f_a}) \vee R\_^{fb}(x, \mathbf{t_a}) \leftarrow Q\_^{fb}(x, \mathbf{t}^\star), R\_^{fb}(x, \mathbf{f_a}), x \neq null.$

$Q\_^{fb}(x, \mathbf{f_a}) \vee R\_^{fb}(x, \mathbf{t_a}) \leftarrow Q\_^{fb}(x, \mathbf{t}^\star),\ not\ R(x), x \neq null.$

$Q\_^{fb}(x, \mathbf{t}^\star) \leftarrow Q\_^{fb}(x, \mathbf{t_a}).$

$Q\_^{fb}(x, \mathbf{t}^\star) \leftarrow Q(x).$

$Q\_^{fb}(x, \mathbf{t}^{\star\star}) \leftarrow Q\_^{fb}(x, \mathbf{t}^\star),\ not\ \ Q\_^{fb}(x, \mathbf{f_a}).$

$R\_^{fb}(x, \mathbf{t}^\star) \leftarrow R\_^{fb}(x, \mathbf{t_a}).$

$R\_^{fb}(x, \mathbf{t}^\star) \leftarrow R(x).$

$R\_^{fb}(x, \mathbf{t}^{\star\star}) \leftarrow R\_^{fb}(x, \mathbf{t}^\star),\ not\ \ R\_^{fb}(x, \mathbf{f_a}).$ □

Different strategies can be used to process atoms and propagate bindings. The process of passing bindings is called *sideways information passing strategies* (SIPS) [8]. Any SIP strategy has to ensure that all of the body and head atoms are processed. We follow the strategy adopted in [29], which is implemented in DLV. According to it, only EDB predicates bind new variables, i.e. variables that do not carry a binding already. As an illustration, suppose we have the adorned predicate P^{fbf} and the following rule:

$$P(x, y, z) \vee T(x, y) \leftarrow R(z), S(x, z),$$

R being a EDB predicate. The adorned rule is:

$$P^{fbf}(x, y, z) \vee T^{fb}(x, y) \leftarrow R(z), S^{fb}(x, z).$$

Notice that variable z is free according to the adorned predicate P^{fbf}. However, the EDB atom $R(z)$ binds this variable, and propagates this binding to atom $S(x, z)$, where variable z becomes *bound* producing the adorned predicate S^{fb}.

Furthermore, when an adorned predicate is processed on a disjunctive rule, only the atoms associated with the adorned predicate produce new bindings. Other head atoms only receive bindings but cannot produce new ones. As an illustration, the adorned predicate $S\_^{bb}$ processed in Rule (5.1) produces Rule (5.2):

$$S\_(x, \mathbf{f_a}) \vee Q\_(x, y, \mathbf{t_a}) \leftarrow S\_(x, \mathbf{t}^{\star}), Q\_(x, y, \mathbf{f_a}), x \neq \mathit{null}. \tag{5.1}$$

$$S\_^{bb}(x, \mathbf{f_a}) \vee Q\_^{bfb}(x, y, \mathbf{t_a}) \leftarrow S\_^{bb}(x, \mathbf{t}^{\star}), Q\_^{bfb}(x, y, \mathbf{f_a}), x \neq \mathit{null}, \tag{5.2}$$

that is, only variable x is bound in atom $Q\_(x, y, \mathbf{t_a})$, but y stays free.

The next step is the *generation of magic rules*; those that simulate a top-down evaluation of the query. They are generated for each rule of the adorned program. The generation differs for disjunctive and non-disjunctive adorned rules.

In the case of non-disjunctive adorned rules, for each adorned atom P^A in the body of an adorned rule, a magic rule is generated as follows:

(a) The head of the magic rule becomes the magic version of P^A, i.e. an atom with predicate symbol $\mathit{magic\_}P^A$, from which all the variables labelled with f in A are deleted.

(b) The atoms in the body of the magic rule become the magic version of the adorned rule head, followed by the atoms (if any) that produced bindings on atom P^A.

As an illustration, considering the adorned atom $S\_^{fb}(x, \mathbf{t_a})$ for the adorned rule:

$$S\_^{fb}(x, \mathbf{t}^{\star}) \leftarrow S\_^{fb}(x, \mathbf{t_a}),$$

the magic rule is:

$$\mathit{magic\_S}\_^{fb}(\mathbf{t_a}) \leftarrow \mathit{magic\_S}\_^{fb}(\mathbf{t}^{\star}).$$

Now, consider the adorned rule:

$$P^{fbf}(x, y, z) \leftarrow R(z), S^{fb}(x, z), T^{b}(z),$$

where there are three atoms in the body, but only two have an adorned predicate. The corresponding magic rule for the adorned atom $S^{fb}(x, z)$ is:

$$\mathit{magic\_S}^{fb}(z) \leftarrow \mathit{magic\_P}^{fbf}(y), R(z).$$

Notice, that the magic rule contains atom $R(z)$, which is added because it bound the variable z on atom $S(x, z)$. If this atom is not introduced in the magic rule, this rule becomes unsafe. The magic rule for $T^{b}(z)$ is:

$$\mathit{magic\_T}^{b}(z) \leftarrow \mathit{magic\_P}^{fbf}(y), R(z).$$

In the case of disjunctive adorned rules, first intermediate non-disjunctive rules are generated. This is achieved by moving, one at a time, head atoms into the bodies of rules. Next, magic rules are generated as described for non-disjunctive rules. For instance, for the rule:

$$S_{\_}^{fb}(x, \mathbf{f_a}) \vee Q_{\_}^{fb}(x, \mathbf{t_a}) \leftarrow S_{\_}^{fb}(x, \mathbf{t}^{\star}), Q_{\_}^{fb}(x, \mathbf{f_a}), x \neq \mathit{null}, \tag{5.3}$$

two non-disjunctive rules are generated by moving atoms $S_{\_}^{fb}(x, \mathbf{f_a})$ and $Q_{\_}^{fb}(x, \mathbf{t_a})$, one at a time, into the body, obtaining the rules:

$$S_{\_}^{fb}(x, \mathbf{f_a}) \leftarrow Q_{\_}^{fb}(x, \mathbf{t_a}), S_{\_}^{fb}(x, \mathbf{t}^{\star}), Q_{\_}^{fb}(x, \mathbf{f_a}), x \neq \mathit{null}, \tag{5.4}$$

$$Q_{\_}^{fb}(x, \mathbf{t_a}) \leftarrow S_{\_}^{fb}(x, \mathbf{f_a}), S_{\_}^{fb}(x, \mathbf{t}^{\star}), Q_{\_}^{fb}(x, \mathbf{f_a}), x \neq \mathit{null}. \tag{5.5}$$

The magic rules for Rule (5.4) are:

(1) $\mathit{magic\_Q\_}^{fb}(\mathbf{t_a}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{f_a})$.,

(2) $\mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{f_a})$., and

(3) $\mathit{magic\_Q\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{f_a})$.

In addition, the magic version of the *Ans* predicate from the adorned query rule is also generated at this step. This magic atom is called the *magic seed* atom. For the adorned query rule:

$$\mathit{Ans}^{f}(x) \leftarrow S\_^{fb}(x, \mathbf{t}^{\star\star}),$$

the magic seed atom is $\mathit{magic\_Ans}^{f}$.

The magic rules for the adorned Program 5.1 are:

Program 5.2

$\mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star\star}) \leftarrow \mathit{magic\_Ans}^{f}$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{t_a}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{f_a})$.

$\mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{f_a})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{f_a})$.

$\mathit{magic\_S\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{t_a})$.

$\mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{t_a})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{t_a})$.

$\mathit{magic\_S\_}^{fb}(\mathbf{t_a}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{t_a}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{t}^{\star})$.

$\mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star\star})$.

$\mathit{magic\_S\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star\star})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{t}^{\star\star})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{t}^{\star\star})$.

$\mathit{magic\_R\_}^{fb}(\mathbf{t_a}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{f_a})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{f_a})$.

$\mathit{magic\_R\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_Q\_}^{fb}(\mathbf{f_a})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_R\_}^{fb}(\mathbf{t_a})$.

$\mathit{magic\_Q\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_R\_}^{fb}(\mathbf{t_a})$.

$\mathit{magic\_R\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_R\_}^{fb}(\mathbf{t_a})$.

$\mathit{magic\_R\_}^{fb}(\mathbf{t_a}) \leftarrow \mathit{magic\_R\_}^{fb}(\mathbf{t}^{\star})$.

$\mathit{magic\_R\_}^{fb}(\mathbf{t}^{\star}) \leftarrow \mathit{magic\_R\_}^{fb}(\mathbf{t}^{\star\star})$.

$\mathit{magic\_R\_}^{fb}(\mathbf{f_a}) \leftarrow \mathit{magic\_R\_}^{fb}(\mathbf{t}^{\star\star})$.

$\mathit{magic\_Ans}^{f}$. □

The last phase is the *modification step*, where magic atoms constructed in the generation stage are included in the body of adorned rules. Thus, for each adorned rule, the magic version of its head is inserted into the body. For instance, the magic versions of the head atoms $S\_^{fb}(x, \mathbf{f_a})$, and $Q\_^{fb}(x, \mathbf{t_a})$ in Rule (5.3) are: $magic\_S\_^{fb}(\mathbf{f_a})$ and $magic\_Q\_^{fb}(\mathbf{t_a})$ respectively, which are inserted into the body of Rule (5.3) generating the modified rule:

$$S\_^{fb}(x, \mathbf{f_a}) \vee Q\_^{fb}(x, \mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}), magic\_Q\_^{fb}(\mathbf{t_a}), S\_^{fb}(x, \mathbf{t}^\star), Q\_^{fb}(x, \mathbf{f_a}), x \neq null.$$

In modified rules the rest of the adornments are now deleted. Hence, the modified rule for Rule (5.3) becomes:

$$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}), magic\_Q\_^{fb}(\mathbf{t_a}), S\_(x, \mathbf{t}^\star), Q\_(x, \mathbf{f_a}), x \neq null.$$

Therefore, the modified rules for the adorned Program 5.1 are:

Program 5.3

$Ans(x) \leftarrow magic\_Ans^f, S\_(x, \mathbf{t}^{\star\star})$.

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}), magic\_Q\_^{fb}(\mathbf{t_a}), S\_(x, \mathbf{t}^\star), Q\_(x, \mathbf{f_a}), x \neq null.$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}), magic\_Q\_^{fb}(\mathbf{t_a}), S\_(x, \mathbf{t}^\star),\ not\ Q(x), x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{f_a}), magic\_R\_^{fb}(\mathbf{t_a}), Q\_(x, \mathbf{t}^\star), R\_(x, \mathbf{f_a}), x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{f_a}), magic\_R\_^{fb}(\mathbf{t_a}), Q\_(x, \mathbf{t}^\star),\ not\ R(x), x \neq null.$

$S\_(x, \mathbf{t}^\star) \leftarrow magic\_S\_^{fb}(\mathbf{t}^\star), S\_(x, \mathbf{t_a}).$ $\quad S\_(x, \mathbf{t}^\star) \leftarrow magic\_S\_^{fb}(\mathbf{t}^\star), S(x).$

$Q\_(x, \mathbf{t}^\star) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^\star), Q\_(x, \mathbf{t_a}).$ $\quad Q\_(x, \mathbf{t}^\star) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^\star), Q(x).$

$R\_(x, \mathbf{t}^\star) \leftarrow magic\_R\_^{fb}(\mathbf{t}^\star), R\_(x, \mathbf{t_a}).$ $\quad R\_(x, \mathbf{t}^\star) \leftarrow magic\_R\_^{fb}(\mathbf{t}^\star), R(x).$

$S\_(x, \mathbf{t}^{\star\star}) \leftarrow magic\_S\_^{fb}(\mathbf{t}^{\star\star}), S\_(x, \mathbf{t}^\star),\ not\ S\_(x, \mathbf{f_a}).$

$Q\_(x, \mathbf{t}^{\star\star}) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^{\star\star}), Q\_(x, \mathbf{t}^\star),\ not\ Q\_(x, \mathbf{f_a}).$

$R\_(x, \mathbf{t}^{\star\star}) \leftarrow magic\_R\_^{fb}(\mathbf{t}^{\star\star}), R\_(x, \mathbf{t}^\star),\ not\ R\_(x, \mathbf{f_a}).$ $\square$

Notice that in the modified rules only the magic atoms keep adornments.

Definition 5.1 For a program Π with a set PC of program constraints, the final rewritten program produced by the MS is denoted by $\mathcal{MS}^{\leftarrow}(\Pi)$ and consists of the magic rules, the modified rules, and PC. $\Box$

Notice that, since in the rewritten program only the magic atoms have adornments, the program constraints can be added as they come to the program without any processing.

The rewritten version of the program in Example 5.1, $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$, consists of the magic rules in Program 5.2, the modified rules in Program 5.3, and the program constraint: $\leftarrow Q\_(x, \mathbf{t_a}), Q\_(x, \mathbf{f_a})$. Program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$ has the following two stable models:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{\mathit{magic\_Ans}^f, \underline{S(a)}, \underline{S\_(a, \mathbf{t}^\star)}, \mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star\star}), \mathit{magic\_S\_}^{fb}(\mathbf{f_a}), \mathit{magic\_S\_}^{fb}(\mathbf{t_a}),\\
& \mathit{magic\_S\_}^{fb}(\mathbf{t}^\star), \mathit{magic\_Q\_}^{fb}(\mathbf{f_a}), \mathit{magic\_Q\_}^{fb}(\mathbf{t_a}), \mathit{magic\_Q\_}^{fb}(\mathbf{t}^\star), \mathit{magic\_R\_}^{fb}(\mathbf{f_a}),\\
& \mathit{magic\_R\_}^{fb}(\mathbf{t_a}), \underline{Q\_(a, \mathbf{t_a})}, \underline{S\_(a, \mathbf{t}^{\star\star})}, \underline{Q\_(a, \mathbf{t}^\star)}, \underline{R\_(a, \mathbf{t_a})}, \underline{Ans(a)}\},\\
\mathcal{M}_2 \;=\; & \{\mathit{magic\_Ans}^f, \underline{S(a)}, \underline{S\_(a, \mathbf{t}^\star)}, \mathit{magic\_S\_}^{fb}(\mathbf{t}^{\star\star}), \mathit{magic\_S\_}^{fb}(\mathbf{f_a}), \mathit{magic\_S\_}^{fb}(\mathbf{t_a}),\\
& \mathit{magic\_S\_}^{fb}(\mathbf{t}^\star), \mathit{magic\_Q\_}^{fb}(\mathbf{f_a}), \mathit{magic\_Q\_}^{fb}(\mathbf{t_a}), \mathit{magic\_Q\_}^{fb}(\mathbf{t}^\star), \mathit{magic\_R\_}^{fb}(\mathbf{f_a}),\\
& \mathit{magic\_R\_}^{fb}(\mathbf{t_a}), \underline{S\_(a, \mathbf{f_a})}\}.
\end{aligned}$$

Since there are no ground Ans-atoms in common, from the original program there are no answers to $\mathcal{Q} : Ans(x) \leftarrow S\_(x, \mathbf{t}^{\star\star})$, which is now expressed as $Ans(x) \leftarrow \mathit{magic\_Ans}^f, S\_(x, \mathbf{t}^{\star\star})$ in the rewritten program.

For the rewritten program only models that are relevant to answer the query are computed. Furthermore, these are partially computed, i.e. they can be extended to stable models of the original program, without considering the magic predicates, which are auxiliary predicates that are used to direct the course of query evaluation.

For instance, model $\mathcal{M}_1$ of program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$ is a subset of the stable models $\mathcal{M}_1$ and $\mathcal{M}_2$ in Example 5.1; and model $\mathcal{M}_3$ is a subset of stable models $\mathcal{M}_3$ and $\mathcal{M}_4$ (without considering magic predicates). Instead of having four stable models as the original program, the rewritten program has only two stable models. In addition, the unique database predicates that are instantiated are the ones related to the query, i.e. Q, and R, in this case via the ICs. For the same reason, program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$ contains rules related to predicates S, Q, R of the original repair program (plus the magic rules), but no rules for predicates T, W, which are not relevant to the query.

Even though in [47] it was shown that for general disjunctive programs with program constraints the MS technique does not always produce an equivalent rewritten program (completeness may be lost), we claim that for the disjunctive repair programs with program constraints as introduced in Definition 4.2, this MS methodology is both sound and complete. As a consequence, the rewritten program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$, and the original repair program $\Pi(D, IC, \mathcal{Q})$ are query equivalent under both brave and cautious reasoning.[1]

Theorem 5.1 Given a database instance D, a set IC of UICs, RICs, and NNCs of the forms (2.3), (2.4) and (2.6), respectively, and a possibly partially ground query $\mathcal{Q}$, program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q})) \equiv_{\mathcal{Q}} \Pi(D, IC, \mathcal{Q})$ under both the brave and cautious semantics. $\Box$

Intuitively, we first prove that the rewritten program $\mathcal{MS}(\Pi^{-}(D, IC, \mathcal{Q}))$ produced by MS, which is applied to a program without program constraints, contains all the

[1]As defined in chapter 2, two programs Π_1 and Π_2 are bravely (resp. cautiously) equivalent wrt a query Q, denoted $\Pi_1 \equiv_{Q} \Pi_2$, if for any set F of facts, brave (resp. cautious) answers to Q from the program $\Pi_1 \cup F$ are the same as the brave (resp. cautious) answers to Q from $\Pi_2 \cup F$.

rules that are needed to check the program constraints for the predicates in program $\mathcal{MS}(\Pi^{-}(D, IC, \mathcal{Q}))$. Therefore, the final program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$ (with program constraints) has only coherent models.

To show soundness of the MS method we need to prove that for each stable model of program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$, there exists a coherent stable model of program $rel(\mathcal{Q}, \Pi^{-}(D, IC, \mathcal{Q}))$ (cf. Definition 5.2 below) that contains a subset of the rules of program $\Pi^{-}(D, IC, \mathcal{Q})$, those that will be used to compute the query $\mathcal{Q}$. Notice that a correspondence between such models can be established by considering non-magic atoms only. Therefore, we need to restrict the models to those atoms.

Finally, to prove the completeness of MS, we have to show that the answers to a query $\mathcal{Q}$ obtained with MS are all the consistent answers to $\mathcal{Q}$.

Now we introduce some definitions and obtain some technical results that are needed for the proof of Theorem 5.1.

Definition 5.2 [29] Given a set S of ground rules of a program Π, $R(S)$ denotes the set $\{r \in \mathit{ground}(\Pi) \mid \exists r' \in S, \exists q \in B(r') \cup H(r') \text{ such that } q \in H(r)\}$.[2] Next, $rel(\mathcal{Q}, \Pi)$ is the least fixed point of the following sequence $rel_0(\mathcal{Q}, \Pi) = \{r \in \mathit{ground}(\Pi) \mid \exists\, q \in \mathcal{Q}, \text{ and } q \cap H(r)\}$, and $rel_{i+1}(\mathcal{Q}, \Pi) = R(rel_i(\mathcal{Q}, \Pi))$, for $i \geq 0$. □

Thus, for a given query $\mathcal{Q}$ and a program Π, program $rel(\mathcal{Q}, \Pi)$ has a subset of the rules of $\mathit{ground}(\Pi)$, those that will be used in the computation of $\mathcal{Q}$. The existence of a fix point for the sequence $rel_i(\mathcal{Q}, \Pi)$ can be guaranteed since the number of rules in repair programs is finite, and also the database domain. We are assuming that the queries are significant, in the sense that they involves database predicates appearing in the repair programs.

[2]As a reminder, $B(r)$ and $H(r)$ denote, respectively, the body and the head of a rule r.

Definition 5.3 [34, 35] For every program Π composed by a finite collection of rules of the form (2.1), there is a marked directed graph $\mathcal{G}(\Pi) = (V, E)$, called the *predicate dependency graph* of Π, which is constructed as follows: each predicate of Π is a node in N, and there is an edge (P_i, P_j) in E from P_i to P_j if there is a rule r such that P_i and P_j occur in the body, respectively, in the head of r. Such an arc is marked if P_i occurs under negation. $\square$

Then, program Π is *stratified* (cf. Definition 4.5) if $\mathcal{G}(\Pi)$ has no cycle with a marked arc, otherwise it is *unstratified*. Moreover, an *odd* cycle in $\mathcal{G}(\Pi)$ is a cycle comprising an odd number of marked arcs.

By personal communication with the authors of [29, 35], we know that the following lemmas, which are a combination of results presented in [29] and [35], have been proven but not published yet.

Lemma 5.1 Given a disjunctive possibly unstratified Datalog program Π, where negation is involved only in even cycles, for every stable model M' of $\mathcal{MS}(\Pi, \mathcal{Q})$, there exists a stable model M of $\mathit{rel}(\mathcal{Q}, \Pi)$, such that $M = M'[\mathit{rel}(\mathcal{Q}, \Pi)]$.[3] $\square$

Lemma 5.2 Given a disjunctive possibly unstratified Datalog program Π, where negation is involved only in even cycles, for every stable model M of $\mathit{rel}(\mathcal{Q}, \Pi)$, there exists a stable model M' of $\mathcal{MS}$ $(\Pi, \mathcal{Q})$, such that $M = M'[\mathit{rel}(\mathcal{Q}, \Pi)]$. $\square$

Notice that the correspondence between the stable models of the magic rewritten program $\mathcal{MS}(\Pi, \mathcal{Q})$, and the stable models of program $\mathit{rel}(\mathcal{Q}, \Pi)$ is established by focusing on non-magic atoms only. This is achieved in Lemmas 5.1 and 5.2 through the condition $M = M'[\mathit{rel}(\mathcal{Q}, \Pi)]$ (cf. Definition 4.3).

[3]As a reminder of Definition 4.3, given a model M of a program Π and a predicate symbol P, $M[P]$ denotes the set of atoms in M whose predicate symbol is P, and $M[\Pi]$ is the set of atoms in M whose predicate symbol appears in the head of some rule in program Π.

We will use Lemmas 5.1 and 5.2 to prove that there exists a correspondence between the stable models of the rewritten program $\mathcal{MS}(\Pi^-(D, IC, \mathcal{Q}))$ produced by our MS, and the stable models of the program $\Pi^-(D, IC, \mathcal{Q})$ restricted to the atoms involved in the computation of the query. In order to use Lemmas 5.1 and 5.2, we need to show that negation does not occur in odd cycles in the repair programs, which, without the program constraints, are locally stratified (cf. Proposition 4.3). This is proved in Proposition 5.1.

Proposition 5.1 For a disjunctive repair program Π, without considering the program constraints, negation is not involved in odd cycles.

Proof: The repair program Π can be rewriten as follows: for every predicate P in Π replace: (a) $P\_(\bar{x}, \mathbf{t_a})$ by $P_{\mathbf{t_a}}(\bar{x})$. (b) $P\_(\bar{x}, \mathbf{f_a})$ by $P_{\mathbf{f_a}}(\bar{x})$. (c) $P\_(\bar{x}, \mathbf{t}^\star)$ by $P_{\mathbf{t}^\star}(\bar{x})$. (d) $P\_(\bar{x}, \mathbf{t}^{\star\star})$ by $P_{\mathbf{t}^{\star\star}}(\bar{x})$. The result follows from the fact that the predicate dependency graph $\mathcal{G}(\Pi')$ does not contain odd cycles. □

In order to show that the program $\mathcal{MS}(\Pi^-(D, IC, \mathcal{Q}))$ has all the rules needed to check the program constraints, we need to introduce some concepts.

Definition 5.4 For a given database instance D, set IC of ICs, and query $\mathcal{Q}$, we denote with $PC_{\mathcal{Q}}$ the set of program constraints of the form $\leftarrow P\_(\bar{x}, \mathbf{t_a}), P\_(\bar{x}, \mathbf{f_a})$ taken from the set PC of program constraints of program $\Pi(D, IC, \mathcal{Q})$, for each predicate P that is connected to a query predicate in $\Pi(\mathcal{Q})$ in the dependency graph $\mathcal{G}(IC)$. Moreover, $PC^\star_{\mathcal{Q}}$ is a program containing only rules of the form $Ans(x) \leftarrow P\_(x, \mathbf{t_a}), P\_(x, \mathbf{f_a})$, for each program constraint in $PC_{\mathcal{Q}}$. □

For the proof of the following propositions, and Theorem 5.1 we simplify the notation as follows: Π denotes the program $\Pi(D, IC,\ \mathcal{Q})$, and Π^- denotes the program

$\Pi^-(D, IC, \mathcal{Q}) := \Pi(D, IC, \mathcal{Q}) \smallsetminus PC$, $\mathcal{MS}(\Pi^-)$ denotes the rewritten program consisting of the magic rules, and the modified rules, and $\mathcal{MS}^{\leftarrow}(\Pi) = \mathcal{MS}(\Pi^-) \cup PC$.

Proposition 5.2 shows that the program $\mathcal{MS}(\Pi^-(D, IC, \mathcal{Q}))$ has all the rules needed to check the program constraints.

Proposition 5.2 For a disjunctive program Π, and the set of program constraints $PC_{\mathcal{Q}}$, the program $\mathit{rel}(PC^\star_{\mathcal{Q}} \ \cup \ \mathcal{Q}, \Pi^-)$ has the same rules as program $\mathit{rel}(\mathcal{Q}, \Pi^-)$.

Proof: It is easy to see that $\mathit{rel}(PC^\star_{\mathcal{Q}} \cup \mathcal{Q}, \Pi^-)$ is equivalent to $\mathit{rel}(PC^\star_{\mathcal{Q}}, \Pi^-) \cup \mathit{rel}(\mathcal{Q}, \Pi^-)$. Hence, it is sufficient to prove that $\mathit{rel}(PC^\star_{\mathcal{Q}}, \Pi^-) \subseteq \mathit{rel}(\mathcal{Q}, \Pi^-)$.

First, if there are no relevant program constraints to answer the query, then program $PC^\star_{\mathcal{Q}}$ has no rules, and $\mathit{rel}(PC^\star_{\mathcal{Q}}, \Pi^-)$ is empty, and we have that $\mathit{rel}(PC^\star_{\mathcal{Q}} \cup \mathcal{Q}, \Pi^-)$ has the same rules as program $\mathit{rel}(\mathcal{Q}, \Pi^-)$. Hence, we focalize on the case where there are program constraints which are relevant to answer the query.

$PC_{\mathcal{Q}}$ is not empty and program $PC^\star_{\mathcal{Q}}$ has at least one rule of the form: $\mathit{Ans}(x) \leftarrow P\_(\bar{a}, \mathbf{t_a}), P\_(\bar{a}, \mathbf{f_a})$. It is easy to see that $\mathit{rel}_0(PC^\star_{\mathcal{Q}}, \Pi^-)$ is composed by the rules of program Π^- whose heads contains either atom $P\_(\bar{a}, \mathbf{t_a})$ or $P\_(\bar{a}, \mathbf{f_a})$.

There are two cases to analyze: first predicate P is a query predicate and, second P is connected to a query predicate in the graph $\mathcal{G}(IC)$.

First, P is a query predicate. Therefore, $\mathit{rel}_0(\mathcal{Q}, \Pi^-)$ is composed by the interpretation rule $P\_(\bar{a}, \mathbf{t}^{\star\star}) \leftarrow P\_(\bar{a}, \mathbf{t}^\star),\ \mathit{not}\ P\_(\bar{a}, \mathbf{f_a})$. Then, $\mathit{rel}_1(\mathcal{Q}, \Pi^-)$ is $\mathit{rel}_0(\mathcal{Q}, \Pi^-)$ plus rules of Π^- with head atoms $P\_(\bar{a}, \mathbf{t}^\star)$ or $P\_(\bar{a}, \mathbf{f_a})$. Then, $\mathit{rel}_2(\mathcal{Q}, \Pi^-)$ is $\mathit{rel}_1(\mathcal{Q}, \Pi^-)$ plus rules of Π^- with head atoms $P\_(\bar{a}, \mathbf{t_a})$ or database facts of the form $P(\bar{a})$. Thus, we have that $\mathit{rel}_0(PC^\star_{\mathcal{Q}}, \Pi^-) \subseteq \mathit{rel}_2(\mathcal{Q}, \Pi^-)$, and as a consequence, $\mathit{rel}(PC^\star_{\mathcal{Q}}, \Pi^-) \subseteq \mathit{rel}(\mathcal{Q}, \Pi^-)$ holds.

Second, P is connected to a query predicate. Therefore, there exists an i such that $\mathit{rel}_i(\mathcal{Q}, \Pi^-)$ has a rule with $P\_(\bar{a}, \mathbf{t_a})$ or $P\_(\bar{a}, \mathbf{f_a})$ in its head, otherwise P is not connected to a query predicate. If P is connected to other predicates, then there exists a j with $j > i$, such that $\mathit{rel}_j(\mathcal{Q}, \Pi^-)$ has a rule with $P\_(\bar{a}, \mathbf{t_a})$ or $P\_(\bar{a}, \mathbf{f_a})$ in its head. Thus, $\mathit{rel}_0(PC^\star_{\mathcal{Q}}, \Pi^-) \subseteq \mathit{rel}_j(\mathcal{Q}, \Pi^-)$, and therefore $\mathit{rel}(PC^\star_{\mathcal{Q}}, \Pi^-) \subseteq \mathit{rel}(\mathcal{Q}, \Pi^-)$ holds. If j does not exists, then we have that $\mathit{rel}_0(PC^\star_{\mathcal{Q}}, \Pi^-) \subseteq \mathit{rel}_i(\mathcal{Q}, \Pi^-)$, and therefore $\mathit{rel}(PC^\star_{\mathcal{Q}}, \Pi^-) \subseteq \mathit{rel}(\mathcal{Q}, \Pi^-)$ holds. □

By Lemmas 5.1 and 5.2 and Proposition 5.1, we know that there exists a correspondence between the stable models of the rewritten program $\mathcal{MS}(\Pi^-)$ produced by MS, and the stable models of the original program, without program constraints, restricted to the atoms involved in the computation of the query (program $\mathit{rel}(\mathcal{Q}, \Pi^-)$). By Proposition 5.2, we know that program $\mathcal{MS}(\Pi^-(D, IC, \mathcal{Q}))$ contains all the rules needed to check the program constraints. Hence, we need to prove that the correspondence between models still holds after introducing the program constraints into the rewritten program $\mathcal{MS}^{\leftarrow}(\Pi) = \mathcal{MS}(\Pi^-) \cup PC$. This is established in the two following propositions, whose proofs use Lemmas 5.1 and 5.2 above.

Proposition 5.3 For every stable model M' of $\mathcal{MS}^{\leftarrow}(\Pi)$, there exists a stable model M of $\mathit{rel}(\mathcal{Q}, \Pi^-)$ such that M is coherent, i.e. it satisfies the set of program constraints $PC_{\mathcal{Q}}$, and $M = M'[\mathit{rel}(\mathcal{Q}, \Pi^-)]$.

Proof: By contradiction, let us assume that there exists a stable model M' of $\mathcal{MS}^{\leftarrow}$ (Π) such that there is no stable model M of $\mathit{rel}(\mathcal{Q}, \Pi^-)$ that is coherent, and $M = M'[\mathit{rel}(\mathcal{Q}, \Pi^-)]$.

M' is in $SM(\mathcal{MS}^{\leftarrow}(\Pi))$. Then, since $\mathcal{MS}^{\leftarrow} = \mathcal{MS}(\Pi^-) \cup PC$, M' is in $SM(\mathcal{MS}$ $(\Pi^-))$. Now, by Lemma 5.1, there exists a model M" of $\mathit{rel}(\mathcal{Q}, \Pi^-)$ such that M" $= M'[\mathit{rel}(\mathcal{Q}, \Pi^-)]$. Since, there are no coherent models of $\mathit{rel}(\mathcal{Q}, \Pi^-)$, M" is incoherent. We have two cases:

(a) M" is incoherent wrt a program constraint d in $PC \smallsetminus PC_{\mathcal{Q}}$. We know that M" is a model of $\mathit{rel}(\mathcal{Q}, \Pi^-)$. By Proposition 5.2, M" is a model of $\mathit{rel}(PC^\star_{\mathcal{Q}} \cup \mathcal{Q}, \Pi^-)$. Since d is in $PC \smallsetminus PC_{\mathcal{Q}}$, there is no rule in $\mathit{rel}(PC^\star_{\mathcal{Q}} \cup \mathcal{Q}, \Pi^-)$ defining atoms that are relevant to d. Then, M" has no atoms relevant to d and it cannot be violated. We have reached a contradiction.

(b) M" is incoherent wrt a program constraint d in $PC_{\mathcal{Q}}$, e.g. $d :\leftarrow S\_(x, \mathbf{t_a})$, $S\_(x, \mathbf{f_a})$. We have that $S\_(x, \mathbf{t_a})$, $S\_(x, \mathbf{f_a})$ are in M", and that M" $= M'[\mathit{rel}$ $(\mathcal{Q}, \Pi^-)]$, therefore $S\_(x, \mathbf{t_a})$, $S\_(x, \mathbf{f_a})$ are in M'. But M' satisfies $PC_{\mathcal{Q}}$. We have reached a contradiction. □

Proposition 5.4 For every stable model M of $\mathit{rel}(\mathcal{Q}, \Pi^-)$, such that M is coherent, i.e. it satisfies the set of program constraints $PC_{\mathcal{Q}}$, there exists a stable model M' of $\mathcal{MS}^{\leftarrow}(\Pi)$, such that $M = M'[\mathit{rel}(\mathcal{Q}, \Pi^-)]$.

Proof: M is in $\mathit{SM}(\mathit{rel}(\mathcal{Q}, \Pi^-))$. Then by Lemma 5.2, there exists a stable model M" of $\mathcal{MS}(\Pi^-)$, such that $M = M$"$[\mathit{rel}(\mathcal{Q}, \Pi^-)]$. M is coherent, therefore M" is coherent as well. Then, since M" is coherent wrt $PC_{\mathcal{Q}}$, it will also be a model of $\mathcal{MS}(\Pi^-) \cup PC_{\mathcal{Q}}$. Now, since $\mathcal{MS}(\Pi^-)$ does not have rules for predicates in $PC \smallsetminus PC_{\mathcal{Q}}$, M" is also a model of $\mathcal{MS}^{\leftarrow}(\Pi) = \mathcal{MS}(\Pi^-) \cup PC$. □

Proof of Theorem 5.1: By Proposition 5.3 and 5.4 we know that there exists a correspondence between the stable models of program $\mathcal{MS}^{\leftarrow}(\Pi)$, which contains program constraints, and the coherent stable models of program $\mathit{rel}(\mathcal{Q}, \Pi^-)$.

Thus, in order to prove the soundness and completeness of MS applied to disjunctive programs with program constraints, we just need to prove that program $(\mathit{rel}(\mathcal{Q}, \Pi^-) \cup PC_{\mathcal{Q}})$ is query equivalent to program $\Pi := \Pi(D, IC, \mathcal{Q})$, under both the cautious and brave semantics. Note that we need to add $PC_{\mathcal{Q}}$ to $\mathit{rel}(\mathcal{Q}, \Pi^-)$ since in the previous propositions we refer to coherent models only.

The ground program Π can be split [63] into a bottom program $\Pi_b = \mathit{rel}(\mathcal{Q}, \Pi^-) \cup PC_{\mathcal{Q}}$, and a top program $\Pi_t = \Pi \smallsetminus \Pi_b$, using as a splitting set all the ground atoms with predicates that are related to the query predicates in $\Pi(\mathcal{Q})$.

This implies that the programs can be hierarchically evaluated in the following way: The stable models of program Π are $\mathit{SM}(\Pi) = \bigcup_M \mathit{SM}(M \cup \Pi_t)$, where the Ms are the stable models of Π_b. The results follows from the fact that for each predicate P in query $\mathcal{Q}$, $\mathit{SM}(\Pi)[P] = (\mathit{SM}(\Pi)\ [\mathit{rel}(\mathcal{Q}, \Pi^-)])[P]$, i.e. the stable models of program Π restricted to the query atoms are the same as the models of program Π restricted to the atoms appearing in program $\mathit{rel}(\mathcal{Q}, \Pi^-)$, and next restricted to the query atoms. As a matter of fact, it can be shown that $\mathit{SM}(\Pi)[\mathit{rel}(\mathcal{Q}, \Pi^-)] = \mathit{SM}(\mathit{rel}(\mathcal{Q}, \Pi^-) \cup PC_{\mathcal{Q}})$. $\square$

Theorem 5.1 establishes that out MS technique is sound and complete, and therefore can be used to evaluate queries. In [29, 47] important results on the application of MS in evaluation of benchmark programs are reported. Moreover, in Chapter 9 we report experimental results on the evaluation of queries by using our MS methodology.

This methodology, based on leaving aside the program constraints when the MS technique is applied, adding them at the end, works always in the case of repair programs. There are two reasons for this: First, the rewritten program $\mathcal{MS}(\Pi^-(D, IC, \mathcal{Q}))$ produced by the MS methodology contains all the rules that are necessary to check the satisfiability of the program constraints that are relevant to the query, in the sense that they contain predicates that are connected to the query predicate in the graph $\mathcal{G}(IC)$. More precisely, we have program constraints of the form $\leftarrow P\_(\bar{x}, \mathbf{t_a}), P\_(\bar{x}, \mathbf{f_a})$ in $\Pi(D, IC)$ only when there are rules defining $P\_(\bar{x}, \mathbf{t_a})$ and $P\_(\bar{x}, \mathbf{f_a})$ in $\Pi(D, IC)$. In $\mathcal{MS}(\Pi^-(D, IC, \mathcal{Q}))$, the output of the MS method, we will still find all the rules defining P; then it will be possible to check the satisfiability of the program constraints in the models of $\mathcal{MS}(\Pi^-(D, IC, \mathcal{Q}))$. Second, with MS we obtain a “subset”(without considering the magic atoms) of the stable models of the original program.

Since the rewritten program contains all the rules that are necessary to check the satisfiability of the program constraints, the stable models of the MS program satisfy the program constraints. Furthermore, each of these models of the MS program contain limited extensions for database predicates (including annotations), those that are sufficient to answer the query as well, however each of them can be extended to a stable model of the original program.

More precisely, it is possible to prove that, for every stable model M of program $\mathcal{MS}^{\leftarrow}$ $(\Pi(D,\ IC,\ \mathcal{Q}))$ (without considering the magic atoms), there is a stable model M' of $\Pi(D, IC, \mathcal{Q})$ that extends M in the sense that $M = M$”, where M” is the set of atoms of M' that appear in the head of a rule in (the ground version of) $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$. As a consequence, the stable models of program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))$ are all coherent models, they contain all the atoms needed to answer a query, and they compute the same answers as the models of program $\Pi(D, IC,$ $\mathcal{Q})$ for the given query.

Our approach works for all our repair programs, but it will not necessarily work for more general disjunctive logic program. Sometimes, even if the MS technique is applied to a disjunctive program with program constraints that does not have stable models, the method can produce a program with stable models. This might happen if the query is related with a part of the program which is consistent regarding the program constraint; and the MS method focalizes on that part of the program to answer the query [47].

In addition, our MS methodology might not work for general logic programs that do have stable models. For instance, for the database instance $\{R(a)\}$ and program Π:

$Y(x) \leftarrow S(x).$ $\qquad$ $S(x) \leftarrow R(x),\ not\ P(x).$

$P(x) \leftarrow R(x),\ not\ S(x).$ $\qquad$ $\leftarrow Y(x).$

there is only one stable model $\mathcal{M} = \{R(a), P(a)\}$. But for query $Ans(x) \leftarrow P(x)$, our MS methodology produces a program that has two stable models (shown here without magic atoms): $\mathcal{M}_1 = \{R(a), P(a), \underline{Ans(a)}\}$; and $\mathcal{M}_2 = \{R(a), S(a)\}$, and as a consequence there are no cautious answers to the query even though $\{a\}$ should be an answer to it. This happens because MS does not select the rule $Y(x) \leftarrow S(x)$. Then, when the constraint $\leftarrow Y(x)$, is put back into the program, it is satisfied even though it should not. In our case, when we deal with repair programs, a rule that is relevant to check the satisfiability of a program constraint is never left out of the rewritten program obtained via MS.

5.3 Applying Magic Sets with the DLV System

From Theorem 5.1 we can conclude that magic sets can be applied to the evaluation of disjunctive repair programs with program constraints. However, we do not know of any system that incorporates MS for programs that contain program constraints.

DLV does implement MS for disjunctive programs, but without program constraints [61]. In fact, when a program with constraints is evaluated in DLV with its built-in MS option, DLV returns a warning message notifying that the program contains program constraints, and that MS cannot be applied. As a consequence, the program is evaluated without MS. On the contrary, when the program does not contain constraints, and it is evaluated with the MS option, DLV applies MS internally, without giving access to the rewritten program (to which it would be easy to add the program constraints at the end). As a consequence, the application of MS with DLV for the evaluation of repair programs with program constraints is not straightforward.

In this section we describe how to modify repair programs in order to be able to apply MS directly through DLV. Basically, program constraints are rewritten in such a way that DLV does not recognize them as program constraints, but it is still able to consider, at query time, only coherent models, i.e. models without a same database atom annotated both with both $\mathbf{t_a}$ and $\mathbf{f_a}$.

Definition 5.5 Given a database instance D, and a set IC of ICs, program $\Pi''(D, IC)$ is obtained from $\Pi(D, IC)$ by replacing in it each program constraint of the form $\leftarrow P\_(\bar{x}, \mathbf{t_a}),\ P\_(\bar{x}, \mathbf{f_a})$ by the rule $\ inc \leftarrow P\_(\bar{x}, \mathbf{t_a}),\ P\_(\bar{x}, \mathbf{f_a})$, being inc a new propositional atom. $\Box$

Program $\Pi''(D, IC)$ may have stable models that are not coherent models of the original program; namely those that contain both $P\_(\bar{c}, \mathbf{t_a})$ and $P\_(\bar{c}, \mathbf{f_a})$ for a given predicate P and a constant c. However, those and only those have the atom inc in them.

Example 5.2 (example 5.1 cont.) Program $\Pi(D, IC)$ has one program constraint, for predicate Q. Hence, $\Pi''(D, IC)$ contains the modified program constraint $\ inc \leftarrow Q\_(x, \mathbf{t_a}),\ Q\_(x, \mathbf{f_a})$, and has two additional stable models:

$$\begin{aligned}
\mathcal{M}_1 &= \{T(a), S(a), T\_(a, \mathbf{t}^{\star}), S\_(a, \mathbf{t}^{\star}), \underline{Q\_(a, \mathbf{t_a})}, S\_(a, \mathbf{t}^{\star\star}), Q\_(a, \mathbf{t}^{\star}), \underline{Q\_(a, \mathbf{f_a})},\\
&\qquad W\_(a, \mathbf{t_a}), \underline{inc}, T\_(a, \mathbf{t}^{\star\star}), W\_(a, \mathbf{t}^{\star}), W\_(a, \mathbf{t}^{\star\star})\},\\
\mathcal{M}_2 &= \{T(a), S(a), T\_(a, \mathbf{t}^{\star}), S\_(a, \mathbf{t}^{\star}), \underline{Q\_(a, \mathbf{t_a})}, S\_(a, \mathbf{t}^{\star\star}), Q\_(a, \mathbf{t}^{\star}), \underline{Q\_(a, \mathbf{f_a})},\\
&\qquad T\_(a, \mathbf{f_a}), \underline{inc}\}.
\end{aligned}$$

Both are incoherent stable models. $\Box$

In order to retrieve consistent answers, a ground query $\mathcal{Q}$ has to be translated into $\mathcal{Q} \vee inc$, and evaluated under the cautious semantics (being true in all stable models), which does not require to discard the incoherent models. This is due to the fact that coherent models do not satisfy atom inc, and then they are required to satisfy $\mathcal{Q}$.

Example 5.3 (example 5.2 cont.) The Datalog query $\mathcal{Q}: Ans \leftarrow \ not \ S(a) \vee R(a)$ expressed as a program contains two rules: $Ans \leftarrow \ not \ S\_(a, \mathbf{t}^{\star\star})$ and $Ans \leftarrow R\_(a, \mathbf{t}^{\star\star})$. The answer to $\mathcal{Q}$ is *yes* in program $\Pi(D, IC, \mathcal{Q})$, but becomes *no* when evaluated in $\Pi''(D, IC, \mathcal{Q})$. However, the query $Ans \leftarrow \ not \ S(a) \vee R(a) \vee inc$, which as a program is: $Ans \leftarrow \ not \ S\_(a, \mathbf{t}^{\star\star})$, $Ans \leftarrow R\_(a, \mathbf{t}^{\star\star})$, and $Ans \leftarrow inc$, is *yes* when evaluated in $\Pi''(D, IC, \mathcal{Q})$. □

Notice that this will also work for queries involving only negative ground literals. This is because, since the answer to a query of this class would be *yes* if each stable model of the program contains atom Ans, the incoherent stable models would not affect a positive answer, since they always contain the atom Ans. On the opposite, if the answer is *no*, then it has to exists a coherent stable model that does not contain the atom Ans. Therefore, the incoherent models would not affect the answer to the query. It is illustrated in the example below.

Example 5.4 (example 5.3 cont.) For the Datalog query $\mathcal{Q}: Ans \leftarrow \ not \ S(b)$, $\Pi(\mathcal{Q})$ is $Ans \leftarrow \ not \ S\_(b, \mathbf{t}^{\star\star})$. The answer to $\mathcal{Q}$ is *yes* in program $\Pi(D, IC, \mathcal{Q})$. The answer to query $Ans \leftarrow \ not \ S(b) \vee inc$, which as a program contains two rules: $Ans \leftarrow \ not \ S\_(b, \mathbf{t}^{\star\star})$, and $Ans \leftarrow inc$, is *yes* in $\Pi''(D, IC, \mathcal{Q})$. This is because since the atom $S\_(b, \mathbf{t}^{\star\star})$ is not in the coherent stable models, the atom Ans is contained in all of them. Moreover, the atom Ans is in all the incoherent stable models of $\Pi''(D, IC, \mathcal{Q})$.

The Datalog query $\mathcal{Q}: Ans \leftarrow \ not \ S(a)$ expressed as a program is: $Ans \leftarrow \ not \ S\_(a, \mathbf{t}^{\star\star})$. The answer to $\mathcal{Q}$ is *no* in program $\Pi(D, IC, \mathcal{Q})$, because there exists a coherent stable model containing atom $S\_(a, \mathbf{t}^{\star\star})$. The answer to query

$Ans \leftarrow \ not\ S(a) \vee inc$, which as a program contains two rules: $Ans \leftarrow \ not\ S\_(a, \mathbf{t}^{\star\star})$, and $Ans \leftarrow inc$, is also *no* in $\Pi''(D, IC, \mathcal{Q})$, because still there exists a coherent stable model having atom $S\_(a, \mathbf{t}^{\star\star})$. $\square$

The case of queries with variables, e.g. $Ans(x) \leftarrow P\_(x, y, \mathbf{t}^{\star\star})$ is slightly different. It cannot be transformed into $Ans(x) \leftarrow P\_(x, y, \mathbf{t}^{\star\star}) \vee inc$, due to the fact that consistent answers are those contained in the intersection of all the *Ans* relations in the coherent stable models of the program. Clearly, incoherent models, those satisfying *inc*, when intersected with the coherent ones, could make us lose cautious answers. In consequence, for incoherent models, we need to make sufficiently large extensions of the *Ans* predicate. We illustrate this situation in the example below.

Example 5.5 For $\Sigma = \{S(A, B), R(A, B), T(A, B, C)\}$, database instance $D = \{S(a, b),\ S(a, c),\ S(b, c),\ R(a, b),\ R(b, c),\ T(b, c, d),\ T(a, b, c).\}$, set IC: $\forall xyz(S(x, y) \wedge S(x, z) \rightarrow y = z)$, $\forall xyz(R(x, y) \wedge R(x, z) \rightarrow y = z)$, (IND) $\forall xy(S(x, y) \rightarrow R(x, y))$, (RIC) $\forall xy(R(x, y) \rightarrow \exists z T(x, y, z))$, and query $\mathcal{Q}$: $Ans(x) \leftarrow S(x, y), R(x, y)$. Program $\Pi''(D, IC, \mathcal{Q})$ has the following rules:

$S(a, b).\ \ S(a, c).\ \ S(b, c).\ \ R(a, b).\ \ R(b, c).\ \ T(b, c, d).\ \ T(a, b, c).$

$S\_(x, y, \mathbf{f_a}) \vee S\_(x, z, \mathbf{f_a}) \leftarrow S\_(x, y, \mathbf{t}^\star), S\_(x, z, \mathbf{t}^\star), y \neq z, x \neq null, y \neq null, z \neq null.$

$R\_(x, y, \mathbf{f_a}) \vee R\_(x, z, \mathbf{f_a}) \leftarrow R\_(x, y, \mathbf{t}^\star), R\_(x, z, \mathbf{t}^\star), y \neq z, x \neq null, y \neq null, z \neq null.$

$S\_(x, y, \mathbf{f_a}) \vee R\_(x, y, \mathbf{t_a}) \leftarrow S\_(x, y, \mathbf{t}^\star), R\_(x, y, \mathbf{f_a}), x \neq null, y \neq null.$

$S\_(x, y, \mathbf{f_a}) \vee R\_(x, y, \mathbf{t_a}) \leftarrow S\_(x, y, \mathbf{t}^\star),\ not\ R(x, y), x \neq null, y \neq null.$

$R\_(x, y, \mathbf{f_a}) \vee T\_(x, y, null, \mathbf{t_a}) \leftarrow R\_(x, y, \mathbf{t}^\star),\ not\ aux\_(x, y), x \neq null, y \neq null.$

$aux\_(x, y) \leftarrow T\_(x, y, z, \mathbf{t}^\star),\ not\ T\_(x, y, z, \mathbf{f_a}), x \neq null, y \neq null, z \neq null.$

$T\_(x, y, z, \mathbf{t}^\star) \leftarrow T(x, y, z). \qquad T\_(x, y, z, \mathbf{t}^\star) \leftarrow T\_(x, y, z, \mathbf{t_a}).$

$R\_(x, y, \mathbf{t}^\star) \leftarrow R(x, y). \qquad\quad R\_(x, y, \mathbf{t}^\star) \leftarrow R\_(x, y, \mathbf{t_a}).$

$S\_(x,y,\mathbf{t}^\star) \leftarrow S(x,y).$ $\qquad$ $S\_(x,y,\mathbf{t}^\star) \leftarrow S\_(x,y,\mathbf{t_a}).$

$R\_(x,y,\mathbf{t}^{\star\star}) \leftarrow R\_(x,y,\mathbf{t}^\star),\ not\ R\_(x,y,\mathbf{f_a}).$

$S\_(x,y,\mathbf{t}^{\star\star}) \leftarrow S\_(x,y,\mathbf{t}^\star),\ not\ S\_(x,y,\mathbf{f_a}).$

$T\_(x,y,z,\mathbf{t}^{\star\star}) \leftarrow T\_(x,y,z,\mathbf{t}^\star),\ not\ T\_(x,y,z,\mathbf{f_a}).$

$inc \leftarrow R\_(x,y,\mathbf{t_a}), R\_(x,y,\mathbf{f_a}).$

$Ans(x) \leftarrow S\_(x,y,\mathbf{t}^{\star\star}), R\_(x,y,\mathbf{t}^{\star\star}).$

Program $\Pi''(D, IC, \mathcal{Q})$ has three stable models:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^\star),\\
& S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^\star), T\_(b,c,d,\mathbf{t}^\star),\\
& aux\_(a,b), aux\_(b,c), S\_(a,c,\mathbf{f_a}), R\_(a,b,\mathbf{t}^{\star\star}), R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{t}^{\star\star}),\\
& S\_(b,c,\mathbf{t}^{\star\star}), T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}), \underline{Ans(a)}, \underline{Ans(b)}\},\\
\mathcal{M}_2 \;=\; & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^\star),\\
& S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^\star), T\_(b,c,d,\mathbf{t}^\star),\\
& aux\_(a,b), aux\_(b,c), R\_(a,c,\mathbf{t_a}), R\_(a,b,\mathbf{f_a}), R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{f_a}),\\
& S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), R\_(a,c,\mathbf{t}^\star), T\_(a,c,null,\mathbf{t_a}), R\_(a,c,\mathbf{t}^{\star\star}), T\_(a,c,\\
& null,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}), T\_(a,c,null,\mathbf{t}^{\star\star}), \underline{Ans(a)}, \underline{Ans(b)}\},\\
\mathcal{M}_3 \;=\; & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^\star),\\
& S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^\star), T\_(b,c,d,\mathbf{t}^\star),\\
& aux\_(a,b), aux\_(b,c), \underline{R\_(a,c,\mathbf{t_a})}, R\_(a,b,\mathbf{t}^{\star\star}), R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{f_a}),\\
& S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), R\_(a,c,\mathbf{t}^\star), \underline{R\_(a,c,\mathbf{f_a})}, T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}),\\
& \underline{Ans(b)}, \underline{inc}\}.
\end{aligned}$$

The stable model $\mathcal{M}_3$ is an incoherent model. If we intersect all the models of program $\Pi''(D, IC, \mathcal{Q})$ we obtain the *Ans*-atom $\{Ans(b)\}$, and therefore the answer to $\mathcal{Q}$ is

(b). Nevertheless, since the intersection of coherent models give us the Ans-atoms $\{Ans(a), Ans(b)\}$, the consistent answers to $\mathcal{Q}$ should be $(a), (b)$. $\Box$

A way to solve this problem is as follows: we introduce into the query program rules of the form:

$$Act(x_i) \leftarrow P\_(\bar{x}, \mathbf{t}^\star), \tag{5.6}$$

for each database predicate P and each variable x_i in $\bar{x}$, such that P is connected to a query predicate in the graph $\mathcal{G}(IC)$ (this could also be done for each database predicate P). In this way we are capturing the active domain of the database (or better, the relevant part of it). If the query Q is domain independent [1] or safe [1], this domain will be large enough to answer it correctly. In consequence, we will assume that queries in this section are domain independent or safe.

We also add the rule:

$$Ans(x_1, \ldots, x_n) \leftarrow inc,\ Act(x_1),\ \ldots,\ Act(x_n), \tag{5.7}$$

to the query program, which in the incoherent models trivializes the answer to the query by accepting in its answer set all the possible combinations of values taken from the (relevant) active domain.[4]

Atoms of the form $P\_(\bar{x}, \mathbf{t}^\star)$ are those that become true during the repair process, i.e. the database facts and the atoms that are made true to restore consistency of ICs. Moreover, notice that if *null* is not in the database domain, but it is introduced when restoring consistency of a RIC, then it will be captured by an atom of the form $P\_(\bar{x}, \mathbf{t}^\star)$. Therefore, these are the ones allowing to give a larger extension for the Ans predicate.

[4]It is easy to adapt this methodology to the case when there are types or distinct abstract domains for the attributes involved in the query.

In other words, in the incoherent models, the answer to the query is a large and relevant (to the query) portion of the active domain. When we intersect this portion with the answers from the coherent models, we get only the latter. Of course, for this to work the query has to be domain independent (or safe, a sound syntactic condition for domain independence) [1].

Example 5.6 (example 5.5 cont.) Predicates $\{S, R, T\}$ are all connected in the corresponding graph $\mathcal{G}(IC)$. Therefore, for $\mathcal{Q}$: $Ans(x) \leftarrow S(x,y),\ R(x,y)$, program $\Pi(\mathcal{Q})$ has the following additional rules:

$Act(x) \leftarrow S\_(x,y,\mathbf{t}^\star)$. $\quad Act(y) \leftarrow S\_(x,y,\mathbf{t}^\star)$.

$Act(x) \leftarrow R\_(x,y,\mathbf{t}^\star)$. $\quad Act(y) \leftarrow R\_(x,y,\mathbf{t}^\star)$.

$Act(x) \leftarrow T\_(x,y,z,\mathbf{t}^\star)$. $\quad Act(y) \leftarrow T\_(x,y,z,\mathbf{t}^\star)$.

$Act(z) \leftarrow T\_(x,y,z,\mathbf{t}^\star)$. $\quad Ans(x) \leftarrow inc, Act(x)$.

Program $\Pi''(D, IC, \mathcal{Q})$ has the stable models:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^\star),\\
& S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^\star), T\_(b,c,d,\mathbf{t}^\star),\\
& aux\_(a,b), aux\_(b,c), Act(a), Act(b), Act(c), Act(d), S\_(a,c,\mathbf{f_a}), R\_(a,b,\mathbf{t}^{\star\star}),\\
& R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}), \underline{Ans(a)},\\
& \underline{Ans(b)}\},\\
\mathcal{M}_2 \;=\; & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^\star),\\
& S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^\star), T\_(b,c,d,\mathbf{t}^\star),\\
& aux\_(a,b), aux\_(b,c), Act(a), Act(b), Act(c), Act(d), R\_(a,c,\mathbf{t_a}), R\_(a,b,\mathbf{t}^{\star\star}),\\
& R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{f_a}), S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), R\_(a,c,\mathbf{t}^\star), R\_(a,c,\mathbf{f_a}),\\
& T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}), \underline{inc}, \underline{Ans(a)}, \underline{Ans(b)}, \underline{Ans(c)}, \underline{Ans(d)}\},
\end{aligned}$$

$$\begin{aligned}\mathcal{M}_3 \;=\; & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^\star),\\ & S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^\star), T\_(b,c,d,\mathbf{t}^\star),\\ & aux\_(a,b), aux\_(b,c), Act(a), Act(b), Act(c), Act(d), R\_(a,c,\mathbf{t_a}), R\_(a,b,\mathbf{f_a}),\\ & R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{f_a}), S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), R\_(a,c,\mathbf{t}^\star), T\_(a,c,null,\mathbf{t_a}),\\ & R\_(a,c,\mathbf{t}^{\star\star}), T\_(a,c,null,\mathbf{t}^\star), T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}), T\_(a,c,null,\mathbf{t}^{\star\star}),\\ & Act(null), \underline{Ans(a)}, \underline{Ans(b)}\}.\end{aligned}$$

Now, $\mathcal{M}_2$ is an incoherent model. Nevertheless if we intersect all the coherent models of program $\Pi''(D, IC, \mathcal{Q})$ we obtain the set of *Ans*-atoms $\{Ans(a), Ans(b)\}$, which is a subset of the *Ans*-atoms $\{Ans(a), Ans(b), Ans(c), Ans(d)\}$ on the incoherent model $\mathcal{M}_2$. Therefore, from the intersection of all the models of program $\Pi''(D, IC, \mathcal{Q})$ the consistent answers to $\mathcal{Q}$ are $(a), (b)$, as expected. □

Proposition 5.5 Given a database instance D, a set IC of ICs, and a safe stratified Datalog query $\mathcal{Q}$, let $\Pi''(\mathcal{Q})$ be the same as query program $\Pi(\mathcal{Q})$ plus rules of the form (5.6), for each predicate P that is connected to a query predicate in $\mathcal{G}(IC)$, plus Rule 5.7. It holds $\Pi''(D, IC, \mathcal{Q}) \equiv_{\mathcal{Q}} \Pi(D, IC, \mathcal{Q})$ under the cautious semantics. □

Proof: For the proof, let Ans_{coh} be the set of *Ans*-atoms from the intersection of all the coherent models of program $\Pi''(D, IC, \mathcal{Q})$, and let Ans_{inc} be the set of *Ans*-atoms in the fixed but arbitrary incoherent model M. We need to show that for every incoherent model M, $Ans_{coh} \subseteq Ans_{inc}$.

By contradiction, let us assume that there exist an *Ans*-atom $Ans(\bar{c})$ that is in Ans_{coh}, but is not in M. If $Ans(\bar{c})$ is not in M, then there exists a query predicate R such that atoms $R\_(\bar{d}, \mathbf{t_a})$ and $R\_(\bar{d}, \mathbf{f_a})$ are in M, and therefore $R\_(\bar{d}, \mathbf{t}^{\star\star})$ is false in M. However, since M is incoherent, then program $\Pi''(D, IC, \mathcal{Q})$ has rules of the form $Act(x_i) \leftarrow P\_(\bar{x}, \mathbf{t}^\star)$ for each predicate P and attribute x_i on $\bar{x}$, such that P is connected to a query predicate.

Since atoms of the form $P\_(\bar{x}, \mathbf{t}^\star)$ are the ones that become true in the repair process, then either $P(\bar{f})$ is in M or $P\_(\bar{f}, \mathbf{t_a})$ is in M, and therefore $Act(e)$ will be true in M for each constant e in the attributes of the relevant predicates to compute the query (including the query predicates). Moreover, since $Ans(\bar{c})$ is a cautious answer, it means that the constants in $\bar{c}$ are in the active domain, then rule $Ans(\bar{c}) \leftarrow inc, Act(c_i), \ldots, Act(c_n)$ in the ground version of $\Pi''(D, IC, \mathcal{Q})$ is satisfied and $Ans(\bar{c})$ is true in M. We have reached a contradiction. $\square$

Notice that the methodology for query answering presented above is applicable in particular to queries that are existentially quantified conjunctions of literals, as long as they are safe, i.e. every variable in negative literal also appears in a positive literal. (And then the method immediately applies to conjunctive queries.) A safe conjunctive query is of the form:

$$Ans(\bar{x}) \leftarrow P_1(\bar{x}_1), \ldots, P_m(\bar{x}_m),\ not\ R_{m+1}(\bar{y}_{m+1}), \ldots,\ not\ R_k(\bar{y}_k), \qquad (5.8)$$

with $\bar{x} \subseteq \bigcup_{i=1}^m \bar{x}_i$, and $\bigcup_{j=m+1}^k \bar{y}_j \subseteq \bigcup_{i=1}^m \bar{x}_i$.

However, for this kind of queries, we can optimize the method above, without producing the possibly large active domain, as follows: instead of inserting rules of the form (5.6) to capture the active domain, we introduce into the query program a rule of the form:

$$Ans(\bar{x}) \leftarrow inc,\ P_{i\_}(\bar{x}_i, \mathbf{t}^\star), \ldots, P_{m\_}(\bar{x}_m, \mathbf{t}^\star), \qquad (5.9)$$

where each P_i is a predicate appearing in a positive literal in query $\mathcal{Q}$ of the form (5.8), such that $\bar{x} \cap \bar{x}_i \neq \emptyset$.

Example 5.7 (example 5.5 cont.) For query $\mathcal{Q}$: $Ans(x) \leftarrow S(x, y),\ R(x, y),\ \ not$

$T(x,x,x)$, atoms $S(x,y)$ and $R(x,y)$ are the positive literals of the query, and for both of them $x \cap \{x,y\} = x$ holds. Therefore, program $\Pi(\mathcal{Q})$ has the following rules:

$Ans(x) \leftarrow S\_(x,y,\mathbf{t}^{\star\star}), R\_(x,y,\mathbf{t}^{\star\star}),\ \ not\ T\_(x,x,x,\mathbf{t}^{\star\star}).$

$Ans(x) \leftarrow inc, S\_(x,y,\mathbf{t}^{\star}), R\_(x,y,\mathbf{t}^{\star}).$

Program $\Pi''(D, IC, \mathcal{Q})$ has three stable models:

$$\begin{aligned}
\mathcal{M}_1 \ = \ & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^{\star}),\\
& S\_(a,c,\mathbf{t}^{\star}), S\_(b,c,\mathbf{t}^{\star}), R\_(a,b,\mathbf{t}^{\star}), R\_(b,c,\mathbf{t}^{\star}), T\_(a,b,c,\mathbf{t}^{\star}), T\_(b,c,d,\mathbf{t}^{\star}),\\
& aux\_(a,b), aux\_(b,c), S\_(a,c,\mathbf{f_a}), R\_(a,b,\mathbf{t}^{\star\star}), R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{t}^{\star\star}),\\
& S\_(b,c,\mathbf{t}^{\star\star}), T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}), \underline{Ans(a)}, \underline{Ans(b)}\},\\
\mathcal{M}_2 \ = \ & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^{\star}),\\
& S\_(a,c,\mathbf{t}^{\star}), S\_(b,c,\mathbf{t}^{\star}), R\_(a,b,\mathbf{t}^{\star}), R\_(b,c,\mathbf{t}^{\star}), T\_(a,b,c,\mathbf{t}^{\star}), T\_(b,c,d,\mathbf{t}^{\star}),\\
& aux\_(a,b), aux\_(b,c), R\_(a,c,\mathbf{t_a}), R\_(a,b,\mathbf{t}^{\star\star}), R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{f_a}),\\
& S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), R\_(a,c,\mathbf{t}^{\star}), R\_(a,c,\mathbf{f_a}), T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}),\\
& \underline{inc}, \underline{Ans(a)}, \underline{Ans(b)}\},\\
\mathcal{M}_3 \ = \ & \{S(a,b), S(a,c), S(b,c), R(a,b), R(b,c), T(a,b,c), T(b,c,d), S\_(a,b,\mathbf{t}^{\star}),\\
& S\_(a,c,\mathbf{t}^{\star}), S\_(b,c,\mathbf{t}^{\star}), R\_(a,b,\mathbf{t}^{\star}), R\_(b,c,\mathbf{t}^{\star}), T\_(a,b,c,\mathbf{t}^{\star}), T\_(b,c,d,\mathbf{t}^{\star}),\\
& aux\_(a,b), aux\_(b,c), R\_(a,c,\mathbf{t_a}), R\_(a,b,\mathbf{f_a}), R\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{f_a}),\\
& S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), R\_(a,c,\mathbf{t}^{\star}), T\_(a,c,null,\mathbf{t_a}), R\_(a,c,\mathbf{t}^{\star\star}),\\
& T\_(a,c,null,\mathbf{t}^{\star}), T\_(a,b,c,\mathbf{t}^{\star\star}), T\_(b,c,d,\mathbf{t}^{\star\star}), T\_(a,c,null,\mathbf{t}^{\star\star}), \underline{Ans(a)},\\
& \underline{Ans(b)}\}.
\end{aligned}$$

$\mathcal{M}_2$ is an incoherent model, nevertheless if we intersect all the coherent models of program $\Pi''(D, IC, \mathcal{Q})$, we obtain the set of *Ans*-atoms $\{Ans(a), Ans(b)\}$, which coincides with the *Ans*-atoms $\{Ans(a), Ans(b)\}$ on the incoherent model $\mathcal{M}_2$. Therefore, from the intersection of all the models of program $\Pi''(D, IC, \mathcal{Q})$ the consistent answers to $\mathcal{Q}$ are $(a), (b)$, as expected. $\square$

In Chapter 9 we present an optimized system that implements our MS methodology for disjunctive repair programs with program constraints. Thus, given a repair and a query program, the system returns the MS rewritten program including the necessary program constraints, which is directly evaluated in the DLV system. In this way, all the extra processing of program constraints and queries presented in this section is avoided. However we decided to present the methodology of this section because it is interesting and relevant in itself.

5.4 Selecting and Importing Relevant Predicates

In this section we present a different optimization method that can be used as an alternative to MS to evaluate queries. This method also captures the relevant database predicates to compute a specific query. However, it is achieved by analyzing the relationship between predicates in the ICs and the query predicates, which is captured by the dependency graph in Definition 2.1.

The relevant predicates are used for pruning repair programs. In this manner, the rules for predicates that are relevant to compute a query are kept in the program, but the other rules are eliminated. By eliminating the database facts that will not be involved in the computation of queries, the flow of data between the database system and the reasoning system, that runs the repair programs, is reduced considerably. In Chapter 9 we report experimental results that prove the effectiveness of this method.

Definition 5.6 A predicate P is *relevant* for the consistent answers to a Datalog query $\mathcal{Q}$ wrt IC if P is in a connected component c of graph $\mathcal{G}(IC)$, and there is a predicate P' appearing in $\mathcal{Q}$ and c. $Rel(\mathcal{Q}, IC)$ denotes the set of relevant predicates for the consistent answers to query $\mathcal{Q}$. $\Box$

Proposition 5.6 Let $\Pi(D, IC, \mathcal{Q}) \downarrow \mathcal{Q}$ denotes the same as program $\Pi(D, IC, \mathcal{Q})$ except that the former contains only rules with head atoms for each predicate P, such that P is in $Rel(\mathcal{Q}, IC)$, and database facts of the form $P(\bar{a})$ with $P(\bar{a}) \in D$. It holds that $\Pi(D, IC, \mathcal{Q}) \downarrow \mathcal{Q}$ and $\Pi(D, IC, \mathcal{Q})$ retrieve the same cautious/brave answers to query $\mathcal{Q}$.

Proof: Let Π' denotes the ground version of program $\Pi(D, IC, \mathcal{Q}) \downarrow \mathcal{Q}$. It is easy to see that the ground program $\Pi(D, IC, \mathcal{Q})$ can be split [63] into a bottom program $\Pi_b(D, IC, \mathcal{Q}) = \Pi'$ and a top program $\Pi_t(D, IC, \mathcal{Q}) = \Pi(D, IC, \mathcal{Q}) \smallsetminus \Pi_b(D, IC, \mathcal{Q})$, using as a splitting set all the atoms with a predicate P that is in $Rel(\mathcal{Q}, IC)$. Thus, program $\Pi_b(D, IC, \mathcal{Q})$ only contains rules for predicates that are connected with the query predicates in the graph $\mathcal{G}(IC)$.

Program $\Pi(D, IC, \mathcal{Q})$ can be hierarchically evaluated in the following way: The models of program $\Pi(D, IC, \mathcal{Q})$ are $SM(\Pi(D, IC, \mathcal{Q})) = \bigcup_M SM(M \cup \Pi_t(D, IC, \mathcal{Q}))$, for stable models M of $\Pi_b(D, IC, \mathcal{Q})$. Since program $\Pi_t(D, IC, \mathcal{Q})$ does not have rules whose predicates are related with the query predicates, the extensions for the Ans predicate (which collects the answers to query $\mathcal{Q}$) can be obtained by using only the rules from the bottom program $\Pi_b(D, IC, \mathcal{Q})$. Then, all the stable models of program $\Pi(D, IC, \mathcal{Q})$ contain the extensions of the Ans predicate. Therefore, $\Pi(D, IC, \mathcal{Q}) \downarrow \mathcal{Q}$ and $\Pi(D, IC, \mathcal{Q})$ retrieve the same cautious/brave answers to $\mathcal{Q}$. $\Box$

Example 5.8 (example 5.1 cont.) Figure 5.1 shows the dependency graph $\mathcal{G}(IC)$ for IC : $\forall x(S(x) \rightarrow Q(x))$, $\forall x(Q(x) \rightarrow R(x))$ and $\forall x(T(x) \rightarrow W(x))$. For query $\mathcal{Q}$: $Ans(x) \leftarrow S\_(x, \mathbf{t}^{\star\star})$, the relevant predicates are $\{S, Q, R\}$.

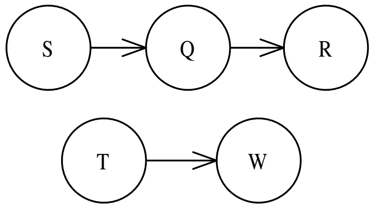


Figure 5.1: Dependency Graph $\mathcal{G}(IC)$ and Relevant Predicates

Therefore, program $\Pi(D, IC, \mathcal{Q})\downarrow\mathcal{Q}$ contains the following rules:

$S(a).$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow S\_(x, \mathbf{t}^\star), Q\_(x, \mathbf{f_a}),\ x \neq null.$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow S\_(x, \mathbf{t}^\star),\ not\ Q(x),\ x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow Q\_(x, \mathbf{t}^\star), R\_(x, \mathbf{f_a}),\ x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow Q\_(x, \mathbf{t}^\star),\ not\ R(x),\ x \neq null.$

$S\_(x, \mathbf{t}^\star) \leftarrow S\_(x, \mathbf{t_a}).\qquad S\_(x, \mathbf{t}^\star) \leftarrow S(x).$

$Q\_(x, \mathbf{t}^\star) \leftarrow Q\_(x, \mathbf{t_a}).\qquad Q\_(x, \mathbf{t}^\star) \leftarrow Q(x).$

$R\_(x, \mathbf{t}^\star) \leftarrow R\_(x, \mathbf{t_a}).\qquad R\_(x, \mathbf{t}^\star) \leftarrow R(x).$

$S\_(x, \mathbf{t}^{\star\star}) \leftarrow S\_(x, \mathbf{t}^\star),\ not\ S\_(x, \mathbf{f_a}).$

$Q\_(x, \mathbf{t}^{\star\star}) \leftarrow Q\_(x, \mathbf{t}^\star),\ not\ Q\_(x, \mathbf{f_a}).$

$R\_(x, \mathbf{t}^{\star\star}) \leftarrow R\_(x, \mathbf{t}^\star),\ not\ R\_(x, \mathbf{f_a}).$

$\leftarrow Q\_(x, \mathbf{t_a}), Q\_(x, \mathbf{f_a}).$

$Ans(x) \leftarrow S\_(x, \mathbf{t}^{\star\star}).$

The stable models are:

$$\begin{aligned}
\mathcal{M}_1 &= \{S(a), S\_(a, \mathbf{t}^\star), Q\_(a, \mathbf{t_a}), S\_(a, \mathbf{t}^{\star\star}), Q\_(a, \mathbf{t}^\star), R\_(a, \mathbf{t_a}), Q\_(a, \mathbf{t}^{\star\star}), R\_(a, \mathbf{t}^\star),\\
&\qquad R\_(a, \mathbf{t}^{\star\star}), \underline{Ans\_(a)}\},\\
\mathcal{M}_2 &= \{S(a), S\_(a, \mathbf{t}^\star), S\_(a, \mathbf{f_a})\}.
\end{aligned}$$

Since there are no ground *Ans*-atoms in common, there are no cautious answers to the query, as expected. However, now a subset of the stable models of program $\Pi(D, IC, \mathcal{Q})$ have been computed. □

It is important to notice that by considering only the relevant predicates to compute consistent answers to queries, there is an important reduction of database facts involved in the computation of the stable models. For instance in Example 5.8, only the database fact $S(a)$ appears in the stable models.

Moreover, it can be proved that there is a one-to-one correspondence between the relevant predicates for a given query and the predicates selected by the MS technique. Thus, both methods presented in this section select the relevant portion of programs to compute a query. However, as we show in Chapter 9, MS produces more gain in terms of execution time of queries, specially for conjunctive queries with free variables and involving more than one database predicate. In Chapter 9 we report experimental results that compare the execution time of queries evaluated with both methodologies.

Let $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))[P]$ denotes the rewritten program produced by MS methodology reduced to the rules having a database predicate P, written as $P_{\_}$, in their heads.

Proposition 5.7 Given a database instance D, a set IC consisting of UICs, RICs, and NNCs, and a query $\mathcal{Q}$, P is in $\mathit{Rel}(\mathcal{Q}, IC)$ iff there is a modified rule r_m in the rewritten program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))[P]$ with $P_{\_}$ in $H(r_m)$, the head of rule r_m.

Proof: We first prove that if P is a relevant predicate to compute a query, then there exists a modified rule in the rewritten program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))[P]$. Next, we prove that if there exists a modified rule in program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))[P]$ then P is relevant.

(a) P is in $\mathit{Rel}(\mathcal{Q}, IC)$, then there is a modified rule r_m in the rewritten program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))[P]$ with $P_{\_}$ in its head.

By contradiction, let us assume that there exists a relevant predicate P, but there is no a rule r_m with $P\_$ in its head. There are two cases. First P is in $\mathcal{Q}$, therefore MS will first adorn the interpretation rule $P\_(\bar{c}, \mathbf{t}^{\star\star}) \leftarrow P\_(\bar{c}, \mathbf{t}^{\star})$, *not* $P\_(\bar{c}, \mathbf{f_a})$. Hence, there is a modified rule with head $P\_(\bar{c}, \mathbf{t}^{\star\star})$ in $\mathcal{MS}^{\leftarrow}$ $(\Pi$ $(D,$ $IC,$ $\mathcal{Q}))[P]$. We have reached a contradiction.

Second, P is not a query predicate, but it is in the same component as a query predicate P'. Therefore, P is directly or transitively connected with P'. Since P' is a query predicate, MS technique will adorn the rules defining P', and will adorn rules with head predicate P at some step i of the adornment process. Therefore there are modified rules with head predicate P in $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))[P]$. We have reached a contradiction.

(b) There is a modified rule r_m in the rewritten program $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, \mathcal{Q}))[P]$ with $P\_$ in its head, then P is in $\mathit{Rel}(\mathcal{Q}, IC)$.

By contradiction, let us assume that there exists a rule r_m with $P\_$ in its head, but P is not in $\mathit{Rel}(\mathcal{Q}, IC)$.

It is easy to see that P cannot be a query predicate, otherwise $P \in \mathit{Rel}(\mathcal{Q}, IC)$ trivially holds. Then, MS selects a rule with predicate P at some step i of the adornment process. First assume $i = 1$. Since P is not a query predicate, the rule being adorned is a disjunctive rule and $P\_$ is in its head. Then, there exists a IC having P either in its antecedent or consequent, and a query predicate P' in its consequent/antecedent. Hence, P is connected with P', and therefore P is in $\mathit{Rel}(\mathcal{Q}, IC)$. We have reached a contradiction.

Assume that P is selected in a step i, with $i > 1$. When $i > 1$, the new rules to be selected are the ones having head predicates selected in the step $i - 1$, which are relevant to compute query answers. For instance, if P is selected in step 2 then, there exists a IC having P either in its antecedent or consequent and a predicate P'' in its consequent/antecedent. Predicate P'' is selected in step 1, and P is connected with P''. Since P'' is selected in step 1, it was connected to a query predicate, therefore P is also connected with a query predicate and P is a relevant predicate. Thus, it is easy to see that for any step i of the adornment process the predicates selected by MS are all relevant to compute queries. Moreover, predicates selected in step i are connected with the predicates selected in step $i-1$, and therefore they are from the same connected component in the graph. We have reached contradiction. □

The selection of relevant database facts performed here differs from the one described in [34], where database repairs are efficiently computed. Here, we are concerning on query answering, but not in computation of repairs.

5.5 Related Work

Optimizations of the process of retrieving consistent answers have been studied and introduced before in the context of data integration. In [34] techniques to efficiently compute and store database repairs are described. Basically, database facts participating in violations of UICs are located and extracted from the database (which does not contain null values). This splits the database in two parts: the affected database, which contains data violating ICs; and the safe database, which stores consistent data. That operation permits to speed up the computation of database repairs, that are computed for the affected part only.

The concepts of safe and affected database wrt UICs is defined in [34] as follows:

Definition 5.7 [34] Consider a database instance D, and a set IC of UICs for a relational database schema $\Sigma = (\mathcal{U}, \mathcal{R} \cup \mathcal{B})$, with *null* $\notin \mathcal{U}$. (a) The facts $P_1(\bar{t}_1)$

and $P_2(\bar{t_2})$, such that $P_1, P_2 \in \mathcal{R}$ and $\bar{t_1}, \bar{t_2}$ have constants in $\mathcal{U}$, are *constraint-bounded* (to each other), if they occur in the same ground (variable-free) constraint $ic_g \in \mathit{ground}(IC)$.[5] (b) $\mathit{facts}(ic_g)$ denotes the set of all facts $P(\bar{t})$ that occur in $ic_g \in \mathit{ground}(IC)$. (c) The *conflict set* for a database instance D is the set of facts $C_D = \{P(\bar{t}) \mid \exists ic_g \in \mathit{ground}(IC) \wedge P(\bar{t}) \in \mathit{facts}(ic_g) \wedge D \not\models ic_g\}$. (d) The *conflict closure* for D, denoted by $C^{\star}_D$, is the least set C with $C_D \subseteq C$ that contains every fact $P(\bar{t})$ constraint-bounded in Σ with some fact $P'(\bar{t}') \in C$. (e) $S_D = D \smallsetminus C^{\star}_D$, and $A_D = D \cap C^{\star}_D$ are the *safe* database and the *affected* database respectively. $\Box$

Intuitively, set $C^{\star}_D$ contains all the facts involved in ICs violations (C_D), and the facts which possibly have to be changed in order to avoid new violations of ICs during the repair process. Note that the conflict set C_D may contain facts that do not belong to the database instance D.

Example 5.9 For $D = \{S(a,b), S(a,c), S(b,c), R(b,c), R(a,b)\}$, primary keys PK_1: $\forall xyz(S(x,y),\ S(x,z) \rightarrow y = z)$; $PK_2 : \forall xyz(R(x,y),\ R(x,z) \rightarrow y = z)$, and IND: $\forall xy(S(x,y) \rightarrow R(x,y))$, the conflict set C_D consists of the tuples that are involved in violations of PK_1, and the IND i.e. $C_D = \{S(a,b), S(a,c), R(a,c)\}$. Here $R(a,c)$ does not belong to D, but $C^{\star}_D = \{S(a,b),\ S(a,c),\ R(a,c),\ R(a,b)\}$. Tuple $R(a,b)$ is constraint-bounded with $R(a,c)$ due to $ic_g : (R(a,c),\ R(a,b) \rightarrow c = b)$. Thus, $S_D = \{S(b,c), R(b,c)\}$ and $A_D = \{S(a,b),\ S(a,c),\ R(a,b)\}$. The database repairs are: $\{S(a,c),\ S(b,c),\ R(a,c), R(b,c)\}$, and $\{S(a,b),\ S(b,c),\ R(a,b), R(b,c)\}$. $\Box$

Having the database split, repairs are computed for the affected database only. In this way, only a subset of the database is involved in the evaluation of repair programs. The idea of this approach is to encode all the repairs into a single database, then after

[5] $\mathit{ground}(IC)$ contains the ground instances of a set IC of ICs.

repairs are computed, a new database instance D' is generated, which contains a copy of the relations in the original database, each of them with an extra attribute called the marked attribute. This attribute is a binary string (containing only 0 and 1) of length n, where n is the number of database repairs for D. Here, 1 in the ith position means that the tuple belongs to the repair i, and 0 means that the tuple is not in the repair i. In this way, the new marked database becomes a compact representation of the database repairs. Consistent answers are computed by first translating Datalog queries into new sentences that consider the new marked attribute; and secondly, evaluating them in the new database instance.

Even though this methodology reduces the amount of data participating in the evaluation of repair programs, at the end it computes database repairs. We do not concentrate on computing repairs, but on efficient computation of consistent answers to queries. Hence, our optimizations are conducted in that direction. We believe that the MS methodology is a better way to achieve this goal (cf. Chapter 9).

5.6 Summary

In this chapter we presented a magic sets methodology that can be applied to our repair programs with program constraints. This technique allows to focalize on a part of the repair programs and facts, those that are relevant to answer the query at hand. It was shown that this methodology is sound and complete when applied to disjunctive repair programs with program constraints.

In order to apply magic sets to repair programs in DLV, a suitable pre-processing of program constraints (if the program contains them) has to be performed. This is due to the fact that currently DLV does not support magic sets for programs with program constraints. In addition, DLV applies magic sets internally, without returning the rewritten program. This implies that it is not possible to add the program constraints later to the magic program.

As an alternative to MS, the evaluation of programs in DLV can also been optimized by selecting the relevant predicates to compute queries, and removing from the repair programs the rules (and database facts) that do not involve relevant predicates. In this manner, programs are smaller that the original ones, and the flow of data between the database system and the reasoning system is decreased.

In Chapter 9 we describe a system that implements our MS methodology to compute consistent answers to queries. The system also implements, as an alternative to MS, the methodology that captures the relevant predicates to compute queries, and uses them to create smaller programs. Moreover, we report experimental results that show that both methods are faster to compute queries than the method which performs a direct evaluation of programs.

Chapter 6

Logic-based Specification of Aggregate Queries

6.1 Introduction

In this chapter we describe how to specify logic programs to compute consistent answers to aggregate queries with *scalar* functions, and aggregate queries with *group-by* statements. The former return a single numerical value by applying an aggregation function like *min, max, count, sum, avg*, to an attribute of a database relation. For instance, the *scalar* aggregate SQL[1] query "`SELECT max(salary) FROM Emp`" returns the maximum value of "salary" in "Emp".

Queries with group-by perform grouping on the values of an attribute or a set of attributes, and return a single value for each group, instead of returning a unique value for the whole relation as the *scalar* aggregate queries do. As an illustration, the SQL query "`SELECT city, sum(sales) FROM Monthly_Sales GROUP BY city`" returns the sum of the "sales" for each "city" from table "Monthly_Sales". This query can also be written as a query program (rule): $Ans(city, sum(sales)) \leftarrow Monthly\_Sales(city, sales)$.

The notion of consistent answer to a *scalar* aggregate query wrt functional dependencies (FDs) was defined in [4]. Intuitively, a consistent answer to an aggregate query is the shortest numerical interval that contains the value of the aggregate query in every possible repair wrt the FDs.

The notion of consistent answer to aggregate queries with *group-by* statements wrt

[1]SQL stands for Structured Query Language.

FDs will be based on this notion of consistent answer given in [4] for *scalar* aggregate queries. Still informally, a tuple of the form $\langle t_1, \ldots, t_m, [a, b]\rangle$ is a consistent answer to an aggregate query with *group-by* statements for the attributes corresponding to $t_1, \ldots, t_m$, if first, the non-aggregate part $\langle t_1, \ldots, t_m\rangle$ is a consistent answer in the usual sense, i.e. it is true in every repair of the database, and secondly, for every repair the aggregate value for the group of attributes values in $t_1, \ldots, t_m$ falls in the numerical interval $[a, b]$.

For instance, for the aggregate query $\mathcal{Q}$:

$$Ans(city, sum(sales)) \leftarrow Monthly\_Sales(city, sales),$$

the tuples $\{(ottawa, [2000, 6000]),\ (montreal, [4000, 7000])\}$ are consistent answers if the cities $ottawa, montreal$ appear in every possible repair, and if for every repair the aggregate value for $sum(sales)$ for $ottawa$ falls in the interval $[2000, 6000]$, and for $montreal$ it falls in the interval $[4000, 7000]$.

As in [4, 5], we restrict our analysis to FDs, but it could be extended to general sets of RIC-acyclic ICs.

In [4, 5], repairs wrt a set of FDs are represented as independent sets in a conflict graph, which is a compact representation of all repairs. Here, we represent repairs as disjunctive logic programs, as done in Chapter 2. By using logic programs, we are able to exploit the capabilities of the DLV system [61] to compute aggregate functions over stable models [36].

The semantics of aggregation under stable models semantics for disjunctive program is investigated in [31, 36], and the implementation of aggregate queries in the DLV system is described in [32]. Currently, the DLV system implements *min, max, count, times, sum*, but not *avg* (average) [37].

The remaining of this chapter is organized as follows: Section 6.2 describes the logic-based specification of repairs for *scalar* aggregate queries. Section 6.3 introduces the notion of consistent answers to aggregate queries with grouping of attributes, together with the logic-based specification of repairs for computing consistent answers. Section 6.4 finalizes this chapter.

6.2 Aggregate Queries with Scalar Functions

A *scalar* aggregate query is of the form:

$$\texttt{SELECT f FROM P}, \tag{6.1}$$

where `f` is one of `min(A), max(A), count(A), sum(A), avg(A)`, and `A` is an attribute of relation P, also `f` can be `count(*)` that returns the number of tuples in relation P.

As a rule, the *scalar* aggregate query will be:

$$Ans(f) \leftarrow P(\bar{x}), \tag{6.2}$$

The following definition of consistent answer is given in [4]. A consistent answer to an aggregate query $\mathcal{Q}$ of the form (6.1) wrt a database instance D, and a set of functional dependencies FD is a minimal numerical interval $I = [a, b]$, such that, for every repair D' of D wrt FD, the numerical value $\mathcal{Q}(D')$ of query $\mathcal{Q}$ in D' belongs to I. The extreme values a, b are called the *greatest lower bound answer* (glb) and the *least upper bound answer* (lub) answers to $\mathcal{Q}$ in D, respectively.

This definition guarantees that the value of the scalar function evaluated in every repair can be found within the most informative interval.

Example 6.1 Consider the database schema $Emp(Name, Salary)$ and FD: $Name \rightarrow$

Salary. The following database instance D violates the FD through the first two tuples:

Emp	Name	Salary
	smith	5000
	smith	8000
	jones	3000

Thus, consistency can be restored by eliminating either tuples *Emp*(*smith*,5000) or *Emp*(*smith*, 8000), therefore there are two repairs:

Emp	Name	Salary
	smith	5000
	jones	3000

Emp	Name	Salary
	smith	8000
	jones	3000

For the *scalar* aggregate query “`SELECT max(Salary) FROM Emp`”, the consistent answer is the interval $[5000, 8000]$, where 5000 is the *glb* answer, and 8000 is the *lub* answer. □

In order to consistently answer *scalar* aggregate queries with logic programs, we first have to generate the repair program $\Pi(D, FD)$ for the corresponding database instance D and set FD of FDs. Notice that since we consider only FDs, the repair program does not have program constraints (cf. Corollary 4.1, Chapter 4).

Instead of expressing aggregate queries in the SQL language, they have to be specified as logic programs. Thus, given an aggregate query $\mathcal{Q}$ of the form (6.1), an aggregate query $\Pi^{\mathcal{A}}(\mathcal{Q})$ is generated by expressing $\mathcal{Q}$ as an aggregate query rule in DLV notation, which, according to the formalism given in [36], is a rule of the form:

$$Ans(w) \ \longleftarrow \ \#f\{x' : P_{\_}(\bar{x}, \mathbf{t}^{\star\star})\} = w, \tag{6.3}$$

where Ans is a new predicate that is not present anywhere else in the repair program, f is the aggregate function in $\mathcal{Q}$, which is applied over variable $x' \in \bar{x}$ of predicate P. The variable x' is in the same position of attribute `A` in (6.1), and w is a variable that will store the aggregate value returned by f in each stable model of the program $\Pi(D, IC)\ \cup \Pi^{\mathcal{A}}(\mathcal{Q})$. Notice that if f is `count(*)` then Rule 6.3 becomes:

$$Ans(w) \longleftarrow \#count\{\bar{x} : P\_(\bar{x}, \mathbf{t}^{\star\star})\} = w, \qquad (6.4)$$

The semantics of aggregation under stable models semantics for disjunctive program present in [31, 36] is defined for *aggregate-stratified* disjunctive programs.

Definition 6.1 [31, 36] A program Π is *aggregate-stratified* if there exists a function $\|\ \|$, called *level mapping*, from the set of predicates in Π to ordinals, such that for each pair P and P' of predicates, occurring in the head and body of a rule r of Π, respectively: (i) if P' appears in an aggregate atom, i.e. an atom over which an aggregate function is computed, then $\|P'\| < \|P\|$, and (ii) if P' occurs in a non-aggregate atom, then $\|P'\| \leq \|P\|$. □

Example 6.2 The following program Π is *aggregate-stratified*, since the level mapping $\|R\| = 1$, $\|P\| = 2$ satisfies the required conditions.
$R(a)$. $R(b)$.
$P(w) \longleftarrow \#count\{x : R(x)\} = w$.
However, if we add the rule $R(x) \longleftarrow P(x)$, then no legal level mapping exists, and therefore the program becomes *aggregate-unstratified.* □

The *aggregate-stratification* condition forbids recursion through aggregates. Notice that our programs are always *aggregate-stratified*, since first repair programs for FDs, that do not have program constraints, are *locally-stratified* (cf. Proposition 4.3, Chapter 4). Moreover, aggregates only appear on rules with the head atom *Ans*, which is not present elsewhere in programs, therefore recursion through aggregates is never introduced.

Example 6.3 (example 6.1 cont.) For query "`SELECT max(Salary) FROM` *Emp*", the program $\Pi^{\mathcal{A}}(\mathcal{Q})$ is $Ans(w) \leftarrow \#max\{y : Emp\_(x, y, \mathbf{t}^{\star\star})\} = w$. Thus, the program $\Pi(D, FD, \mathcal{Q}) := \Pi(D, FD) \cup \Pi^{\mathcal{A}}(\mathcal{Q})$ has the following rules:

$Emp(smith, 5000)$. $Emp(smith, 8000)$. $Emp(jones, 3000)$.

$Emp\_(x, y, \mathbf{f_a}) \vee Emp\_(x, z, \mathbf{f_a}) \leftarrow Emp\_(x, y, \mathbf{t}^{\star}),\ Emp\_(x, z, \mathbf{t}^{\star}),\ y \neq z,\ x \neq null,$

$y \neq null,\ z \neq null.$

$Emp\_(x, y, \mathbf{t}^{\star}) \leftarrow Emp\_(x, y, \mathbf{t_a})$.

$Emp\_(x, y, \mathbf{t}^{\star}) \leftarrow Emp(x, y)$.

$Emp\_(x, y, \mathbf{t}^{\star\star}) \leftarrow Emp\_(x, y, \mathbf{t}^{\star}), not\ Emp\_(x, y, \mathbf{f_a})$.

$Ans(w) \leftarrow \#max\{y : Emp\_(x, y, \mathbf{t}^{\star\star})\} = w$. □

Even though the specification above conforms to the formalism given in [36], and the semantics of such programs returns the correct answers, the program will not run in the DLV system. The reason is that DLV currently presents technical difficulties with aggregation over predicates that are defined by unstratified or disjunctive rules, as in the previous example, where the rule defining atom $Emp\_(x, y, \mathbf{t}^{\star\star})$ involves atom $Emp\_(x, y, \mathbf{f_a})$ which is in the head of a disjunctive rule. In these cases, DLV returns a warning message notifying that the aggregate function cannot be applied on disjunctive or unstratified predicates.

In these cases problems arise during the grounding process of DLV, which is performed before the computation of the stable models. For instance, variable w in rule $Ans(w) \leftarrow \#max\{y : Emp\_(x, y, \mathbf{t}^{\star\star})\} = w$ (cf. Example 6.3) is unbound before the ground version of the program is computed, and DLV would have to compute all possible values for binding it when the ground version of the program is computed. For the functions *max, min* the possible values are among the values taken by variable y, but for other functions, like *sum*, they are not, and therefore grounding could become very difficult.

As a consequence, in order for DLV to answer queries involving functions *max* and *min*, rules have to be modified by inserting an extra argument that binds the aggregation variable. Thus, assuming, without loss of generality, that the last variable in atom $P(\bar{x})$ corresponds to x', i.e. the variable to which the aggregate function f (*max* or *min*) is applied, Rule 6.3 is transformed into:

$$Ans(w) \leftarrow \#f\{x' : P\_(\bar{x}, \mathbf{t}^{\star\star})\} = w,\ P\_(\bar{y}, w, \mathbf{t}^{\star\star}), \tag{6.5}$$

where w is bounded by atom $P\_(\bar{y}, w, \mathbf{t}^{\star\star})$, and $\bar{y}$ are fresh variables for variables in $\bar{x} \smallsetminus \{x'\}$. This solution works because the *max* and *min* are taken inside relation P (for a finite Herbrand domain).

Notice that for the aggregate functions *sum* and *count* it is not possible in general to bind the variable w to a value in a database predicate. Therefore, we are able to compute consistent answers, in DLV system, to aggregate queries involving only the aggregate functions *max* and *min*.

Example 6.4 (example 6.3 cont.) Program $\Pi^{\mathcal{A}}(\mathcal{Q})$ is now: $Ans(w) \leftarrow \#max\{y : Emp\_(x, y, \mathbf{t}^{\star\star})\} = w,\ Emp\_(z, w, \mathbf{t}^{\star\star})$. Program $\Pi(D, FD, \mathcal{Q})$ has two stable models:[2]

[2]Here and in the rest, the models of the programs are displayed without program facts.

$$\begin{array}{rcl}\mathcal{M}_1 & = & \{\mathit{Emp\_}(\mathit{smith}, 5000, \mathbf{t}^\star), \mathit{Emp\_}(\mathit{smith}, 8000, \mathbf{t}^\star), \mathit{Emp\_}(\mathit{jones}, 3000, \mathbf{t}^\star), \\ & & \mathit{Emp\_}(\mathit{smith}, 5000, \mathbf{t}^{\star\star}), \mathit{Emp\_}(\mathit{smith}, 8000, \mathbf{f_a}), \mathit{Emp\_}(\mathit{jones}, 3000, \mathbf{t}^{\star\star}), \\ & & \underline{\mathit{Ans}(5000)}\}, \\ \mathcal{M}_2 & = & \{\mathit{Emp\_}(\mathit{smith}, 5000, \mathbf{t}^\star), \mathit{Emp\_}(\mathit{smith}, 8000, \mathbf{t}^\star), \mathit{Emp\_}(\mathit{jones}, 3000, \mathbf{t}^\star), \\ & & \mathit{Emp\_}(\mathit{smith}, 5000, \mathbf{f_a}), \mathit{Emp\_}(\mathit{smith}, 8000, \mathbf{t}^{\star\star}), \mathit{Emp\_}(\mathit{jones}, 3000, \mathbf{t}^{\star\star}), \\ & & \underline{\mathit{Ans}(8000)}\}.\end{array}$$

The aggregation function returns 5000 as the maximum salary in the first repair, and 8000 in the second one. □

Consistent answers, intervals in this case, are computed by capturing all the values returned by the aggregation function across the models, which can be achieved by running the program $\Pi(D, FD, \mathcal{Q}) := \Pi(D, FD) \cup \Pi^{\mathcal{A}}(\mathcal{Q})$ under the brave semantics [44].

The brave answers returned can be used as database facts for a separate program that contains rules for the aggregate functions *min* and *max*, that once computed, will return the left and right extremes of the (minimal) consistent interval, resp. So, we use the rules

$$\mathit{glb}(w) \leftarrow \#\mathit{min}\{x : \mathit{Ans}(x)\} = w, \mathit{Ans}(w). \tag{6.6}$$

$$\mathit{lub}(w) \leftarrow \#\mathit{max}\{x : \mathit{Ans}(x)\} = w, \mathit{Ans}(w). \tag{6.7}$$

where *Ans* is the same as in Rule 6.5. The consistent answer to $\mathcal{Q}$ is obtained from the unique stable model, and corresponds to the numerical interval $[\mathit{glb}(\bar{c}), \mathit{lub}(\bar{c})]$.

Example 6.5 (example 6.4 cont.) Tuples $\mathit{Ans}(5000)$, $\mathit{Ans}(8000)$ are the *Ans*-atoms from all the stable models of program $\Pi(D, FD, \mathcal{Q})$. They become facts of a new program that contains the Rules 6.6 and 6.7. This new program has one stable model

$\mathcal{M} = \{glb(5000),\ lub(8000)\}$. Therefore, the consistent answer to query "`SELECT max(Salary) FROM Emp`" is the numerical interval $[5000, 8000]$, as expected. □

Algorithm 6.1: CQA-Scalar Aggregate Queries($D, FD, \mathcal{Q}$)

```
**Input**: The database instance $D$, the set $FD$ of FDs, the aggregate query $\mathcal{Q}$
**Output**: consistent answer to query $\mathcal{Q}$
$\Pi(D, FD) := \mathit{GenerateRepairProgram}(D, FD)$;
$\Pi^{\mathcal{A}}(\mathcal{Q}) := \mathit{GenerateAProgram}(\mathcal{Q})$;
$\Pi(D, FD, \mathcal{Q}) := \Pi(D, FD) \cup \Pi^{\mathcal{A}}(\mathcal{Q})$;
$\Pi := \mathit{GenerateNewProgram}()$;
$\Pi := \mathit{ComputeBraveAnswers}(\Pi(D, FD, \mathcal{Q}))$;
Add to $\Pi$ rules:
$glb(w) \leftarrow \#min\{x : Ans(x)\} = w, Ans(w).$
$lub(w) \leftarrow \#max\{x : Ans(x)\} = w, Ans(w).$
$\mathit{ComputeStableModel}(\Pi)$;
**return** $([glb(\bar{c}), lub(\bar{c})])$
```

Algorithm 6.1 computes consistent answers to *scalar* aggregate queries of the form (6.1). The input to the algorithm consists of the database instance D, the set FD of FDs, and the aggregate query $\mathcal{Q}$. It first generates the repair program $\Pi(D, FD)$ and the query program $\Pi^{\mathcal{A}}(\mathcal{Q})$, which are run together in DLV under the brave semantics. In this way all the answers for the aggregate query from every stable model are captured. The brave answers are used as program facts of a second program Π, which also contains Rules 6.6 and 6.7. The consistent interval $[glb(\bar{c}), lub(\bar{c})]$ is obtained from the unique stable model of program Π.

It is important to mention that the introduction of aggregates in disjunctive programs does not increase the intrinsic complexity of the brave and cautious reasoning tasks [31, 36], which are Σ_2^P-complete, and Π_2^P-complete, respectively. Briefly, Σ_2^P stands for NP^{NP}, i.e. the NP problems that are solved with an oracle in NP. On the contrary, Π_2^P stands for co-NP^{NP}, that is the problems whose complement problems are in NP^{NP} [30].

6.3 Aggregate Queries with Group-By Statements

An aggregate query with *group-by* statements is of the form:

$$\texttt{SELECT A}_i, \ldots, \texttt{A}_m, \texttt{f(A) FROM P GROUP BY A}_i, \ldots \texttt{A}_m, \tag{6.8}$$

where $\texttt{A}_i, \ldots \texttt{A}_m, \texttt{A}$ are attributes of relation `P`, and `f` is one of `min(A), max(A), count(A), sum(A), avg(A)`, applied to attribute `A` with $\texttt{A} \cap \{\texttt{A}_i, \ldots \texttt{A}_m\} = \emptyset$.

A query like this can be expressed by means of a logic program Π containing an answer predicate $Ans(x_1, \ldots, x_m, w)$, where the values of w are computed with the aggregation function f, that is parameterized by the values $(x_1, \ldots, x_m)$ and applied to a set of values of attribute `A`. We simply denote this value by $f(x_1, \ldots, x_m)$, so that the answers to the query are of the form $Ans(x_1, \ldots, x_m, f(x_1, \ldots, x_m))$. Typically, such an aggregate query will be based on a conjunctive query, in which case the query program rule will take the form

$$Ans(x_1, \ldots, x_m, w) \longleftarrow P_1(\bar{x}_1), \ldots, P_k(\bar{x}_k), w = f(x_1, \ldots, x_m),$$

with $\{x_1, \ldots, x_m\} \subseteq \bigcup_{j=1}^{k} \bar{x}_j$.

The definition of consistent answer for this kind of queries is inspired by the notion of consistent answer to *scalar* aggregate queries [4].

Definition 6.2 Consider a database instance D, and a set FD of FDs, and an aggregate query with group-by $\mathcal{Q}$ whose answer is given trough an answer predicate $Ans(x_1, \ldots, x_m, w)$ defined by a logic program with aggregation function $f(x_1, \ldots, x_m)$.

A *consistent answer* to query $\mathcal{Q}$ wrt FD is a tuple of the form $\langle t_1, \ldots, t_m, [a, b] \rangle$ such that: (a) $[a, b]$ is a numerical interval. (b) For every repair D' of D wrt FD, $\langle t_1, \ldots, t_m, f(t_1, \ldots, t_m) \rangle$ is an answer to Q in D' and $f(t_1, \ldots, t_m) \in [a, b]$. (c) $[a, b]$ is the shortest interval with the properties above. $\square$

We can see that the non-aggregate part $\langle t_1, \ldots, t_m \rangle$ of a consistent answer for an aggregation query is a consistent answer in the usual non-aggregate sense. The extreme values a, b are called the *greatest lower bound answer* (glb) and the *least upper bound answer* (lub) answers for $\langle t_1, \ldots, t_m \rangle$ in D, respectively. If $a = b$ then the interval can be represented as $[a]$.

Example 6.6 For database schema $Dept(ID, Name, Budget, Year)$ and FD: $ID \rightarrow Name$, the following database instance D violates the FD through the first four tuples.

Dept	ID	Name	Budget	Year
	1	*cs*	5000	2000
	1	*cs*	8000	2001
	1	*math*	3000	2000
	1	*math*	6000	2001
	2	*biol*	3000	2001
	2	*biol*	7000	2002

Consistency can be restored by eliminating either database tuples $\{Dept(1, cs$, 5000, 2000), $Dept(1, cs, 8000, 2001)\}$ or tuples $\{Dept(1, math$, 3000, 2000), $Dept(1$, $math$, 6000, 2001)$\}$, hence there are two repairs:

Dept	ID	Name	Budget	Year
	1	*cs*	5000	2000
	1	*cs*	8000	2001
	2	*biol*	3000	2001
	2	*biol*	7000	2002

Dept	ID	Name	Budget	Year
	1	*math*	3000	2000
	1	*math*	6000	2001
	2	*biol*	3000	2001
	2	*biol*	7000	2002

The answers to the query "`SELECT Year, max(Budget) FROM Dept GROUP BY Year`" from the first and second repairs are: $\{(2000, 5000)$, (2001, 8000), (2002, 7000)$\}$, and $\{$(2000, 3000), (2001, 6000), (2002, 7000)$\}$ respectively. For years 2000 and 2001 there are different answers in the repairs, that is because tuples with these years are inconsistent wrt the FD. However, for year 2002 the answer is the same in both repairs, which happens because the tuple (2, *biol*, 7000, 2002) is consistent wrt the FD. Hence, the consistent answers to $\mathcal{Q}$ are $(2000, [3000, 5000])$, $(2001, [6000, 8000])$, $(2002, [7000])$, i.e. for years 2000 and 2001 it is a minimal numerical interval that contains the value of the aggregation function in every repair, and for year 2002 it is a unique value.

For query "`SELECT Name, max(Budget) FROM Dept GROUP BY Name`" the answers from the first repair are $\{(cs, 8000), (biol, 7000)\}$, and $\{(math$, 6000), $(biol$, 7000)$\}$ from the second one. Hence, the consistent answer to $\mathcal{Q}$ is $(biol, [7000])$, because *biol* belongs to both repairs. However, for departments *cs* and *math* it is not possible to compute a consistent answer, since they do not appear in all the repairs. $\square$

It is important to remark that if there exists a repair where the aggregation function is not defined for a certain group of attributes, then there is no consistent answer for that group of attributes, as illustrated in the previous example.

As it is the case for *scalar* aggregate queries, in order to use logic programs to compute consistent answers to queries with *group-by* statements, we have to generate the repair program $\Pi(D, FD)$, for the database instance and set FD of FDs, and translate the aggregate query into a logic program. Thus, given an aggregate query $\mathcal{Q}$ of the form (6.8), a query program $\Pi^{\mathcal{A}}(\mathcal{Q})$ is generated by expressing $\mathcal{Q}$ as an aggregate rule [36] of the form:

$$Ans(\bar{y}, w) \leftarrow \#f\{x' : P\_(\bar{x}, \mathbf{t}^{\star\star})\} = w,\ P\_(\bar{z}, \bar{y}, w, \mathbf{t}^{\star\star}), \tag{6.9}$$

where Ans is a new predicate that is not present elsewhere in the repair program, f is the aggregate function *max* or *min* in $\mathcal{Q}$, which is applied over variable $x' \in \bar{x}$ of predicate P and corresponds to attribute $\mathtt{A}$, $\bar{y} \subseteq \bar{x}$ is a set of the form $y_i, \ldots, y_m$ and corresponds to the positions of attributes in $\mathtt{A}_i, \ldots, \mathtt{A}_m$ (they are variables in the *group-by* statement of $\mathcal{Q}$), variables in $\bar{z}$ are fresh variables for variables in $\bar{x} \smallsetminus (\bar{y} \cup \{x'\})$, and w is a variable that stores the value returned by f in each stable model of program $\Pi(D, FD) \cup \Pi^{\mathcal{A}}(\mathcal{Q})$. Atom $P\_(\bar{z}, \bar{y}, w, \mathbf{t}^{\star\star})$ binds the variables $\{\bar{y}, w\}$ in each stable model. We assume, without loss of generality, that the variables in atom $P(\bar{x})$ appear in the following order: (1) variables in $\bar{x} \smallsetminus (\bar{y} \cup \{x'\})$, (2) variables in $\bar{y}$, and (3) variable x'.

Example 6.7 (example 6.6 cont.) For query "`SELECT Year, max(Budget) FROM Dept GROUP BY Year`", the program $\Pi(D, FD, \mathcal{Q})$ is:

$Dept(1, cs, 5000, 2000)$. $Dept(1, cs, 8000, 2001)$. $Dept(1, math, 3000, 2000)$.
$Dept(1, math, 6000, 2001)$. $Dept(2, biol, 3000, 2001)$. $Dept(2, biol, 7000, 2002)$.
$Dept\_(x, y, j, w, \mathbf{f_a}) \vee Dept\_(x, z, u, v, \mathbf{f_a}) \leftarrow Dept\_(x, y, j, w, \mathbf{t}^{\star}), Dept\_(x, z, u, v, \mathbf{t}^{\star}),$

$$y \neq z, x \neq null, y \neq null, z \neq null.$$

$Dept\_(x, y, z, w, \mathbf{t}^{\star}) \leftarrow Dept\_(x, y, z, w, \mathbf{t_a}).$

$Dept\_(x, y, z, w, \mathbf{t}^{\star}) \leftarrow Dept(x, y, z, w).$

$Dept\_(x, y, z, w, \mathbf{t}^{\star\star}) \leftarrow Dept\_(x, y, z, w, \mathbf{t}^{\star}),\ not\ Dept\_(x, y, z, w, \mathbf{f_a}).$

$Ans(v, w) \leftarrow \#max\{z : Dept\_(x, y, z, v, \mathbf{t}^{\star\star})\} = w, Dept\_(t, r, w, v, \mathbf{t}^{\star\star}).$

The stable models of program $\Pi(D, FD, \mathcal{Q})$ are:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{Dept\_(1, cs, 5000, 2000, \mathbf{t}^{\star}), Dept\_(1, cs, 8000, 2001, \mathbf{t}^{\star}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{t}^{\star}), Dept\_(1, math, 6000, 2001, \mathbf{t}^{\star}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t}^{\star}), Dept\_(2, biol, 7000, 2002, \mathbf{t}^{\star}),\\
& Dept\_(1, cs, 5000, 2000, \mathbf{f_a}), Dept\_(1, cs, 8000, 2001, \mathbf{f_a}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{t}^{\star\star}), Dept\_(1, math, 6000, 2001, \mathbf{t}^{\star\star}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t}^{\star\star}), Dept\_(2, biol, 7000, 2002, \mathbf{t}^{\star\star}),\\
& \underline{Ans(2000, 3000)}, \underline{Ans(2001, 6000)}, \underline{Ans(2002, 7000)}\},\\
\mathcal{M}_2 \;=\; & \{Dept\_(1, cs, 5000, 2000, \mathbf{t}^{\star}), Dept\_(1, cs, 8000, 2001, \mathbf{t}^{\star}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{t}^{\star}), Dept\_(1, math, 6000, 2001, \mathbf{t}^{\star}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t}^{\star}), Dept\_(2, biol, 7000, 2002, \mathbf{t}^{\star}),\\
& Dept\_(1, cs, 5000, 2000, \mathbf{t}^{\star\star}), Dept\_(1, cs, 8000, 2001, \mathbf{t}^{\star\star}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{f_a}), Dept\_(1, math, 6000, 2001, \mathbf{f_a}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t}^{\star\star}), Dept\_(2, biol, 7000, 2002, \mathbf{t}^{\star\star}),\\
& \underline{Ans(2000, 5000)}, \underline{Ans(2001, 8000)}, \underline{Ans(2002, 7000)}\}.
\end{aligned}$$

Thus, from stable model $\mathcal{M}_1$, the maximum budgets for years $2000, 2001$, and 2002 are $3000, 6000$, and 7000 respectively, because atoms Ans(2000, 3000), Ans(2001, 6000), Ans(2002, 7000) are in $\mathcal{M}_1$. The maximum budgets from stable model $\mathcal{M}_2$

are 5000, 8000, and 7000 respectively, because atoms $Ans(2000,\ 5000)$, $Ans(2001,\ 8000)$, $Ans(2002,\ 7000)$ are in $\mathcal{M}_2$.

For query "`SELECT Name, max(Budget) FROM Dept GROUP BY Name`", the program $\Pi^{\mathcal{A}}(\mathcal{Q})$ is $Ans(y, w) \leftarrow \#max\{z : Dept\_(x, y, z, v, \mathbf{t^{\star\star}})\} = w, Dept\_(t, y, w, r, \mathbf{t^{\star\star}})$, program $\Pi(D,\ FD,\ \mathcal{Q})$ has two stable models:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{Dept\_(1, cs, 5000, 2000, \mathbf{t^\star}), Dept\_(1, cs, 8000, 2001, \mathbf{t^\star}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{t^\star}), Dept\_(1, math, 6000, 2001, \mathbf{t^\star}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t^\star}), Dept\_(2, biol, 7000, 2002, \mathbf{t^\star}),\\
& Dept\_(1, cs, 5000, 2000, \mathbf{t^{\star\star}}), Dept\_(1, cs, 8000, 2001, \mathbf{t^{\star\star}}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{f_a}), Dept\_(1, math, 6000, 2001, \mathbf{f_a}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t^{\star\star}}), Dept\_(2, biol, 7000, 2002, \mathbf{t^{\star\star}}),\\
& \underline{Ans(cs, 8000)}, \underline{Ans(biol, 7000)}\},\\
\mathcal{M}_2 \;=\; & \{Dept\_(1, cs, 5000, 2000, \mathbf{t^\star}), Dept\_(1, cs, 8000, 2001, \mathbf{t^\star}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{t^\star}), Dept\_(1, math, 6000, 2001, \mathbf{t^\star}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t^\star}), Dept\_(2, biol, 7000, 2002, \mathbf{t^\star}),\\
& Dept\_(1, cs, 5000, 2000, \mathbf{f_a}), Dept\_(1, cs, 8000, 2001, \mathbf{f_a}),\\
& Dept\_(1, math, 3000, 2000, \mathbf{t^{\star\star}}), Dept\_(1, math, 6000, 2001, \mathbf{t^{\star\star}}),\\
& Dept\_(2, biol, 3000, 2001, \mathbf{t^{\star\star}}), Dept\_(2, biol, 7000, 2002, \mathbf{t^{\star\star}}),\\
& \underline{Ans(math, 6000)}, \underline{Ans(biol, 7000)}\}.
\end{aligned}$$

Hence, from stable model $\mathcal{M}_1$ the maximum budgets are 8000 and 7000 for departments *cs* and *biol*, respectively ($Ans(cs, 8000), Ans(biol, 7000)$ are in $\mathcal{M}_1$), and from stable model $\mathcal{M}_2$ the maximum budgets are 6000 and 7000 for departments *math* and *biol*, respectively ($Ans(math,\ 6000)$, $Ans(biol, 7000)$ are in $\mathcal{M}_2$). □

In order to compute consistent intervals, we first need to capture the consistent groups of attributes i.e. those that appear in every repair. This is because, consistent answers are computed for the groups of attributes for which the aggregate value is defined in every repair (stable model). However, we cannot obtain the *Ans*-atoms from the intersection of the stable models of program $\Pi(D, FD, \mathcal{Q})$, as we illustrate in the following example.

Example 6.8 (example 6.7 cont.) For query "`SELECT Year, max(Budget) FROM Dept GROUP BY Year`", the only *Ans*-atom that remains in the intersection of the stable models of program $\Pi(D, FD, \mathcal{Q})$ is $Ans(2002,\ 7000)$, therefore we loose information about years 2000 and 2001 which appear in both repairs. □

The consistent group of attributes can be obtained by evaluating program $\Pi(D, FD, \mathcal{Q})$ with the following rule under cautious reasoning [44]:

$$Ans'(\bar{y}) \leftarrow Ans(\bar{y}, w), \tag{6.10}$$

where Ans' is a new predicate that is not present anywhere else in the repair program, $\bar{y}$ are the variables for the attributes in the group-by, and w is the variable that stores the aggregate value, as before.

In this way, the Ans' predicate captures the group of attributes for which there exists an aggregate value in every possible repair.

Example 6.9 (example 6.8 cont.) For query "`SELECT Year, max(Budget) FROM Dept GROUP BY Year`", the Ans'-atoms that remains in the intersection of the stable models of program $\Pi(D, FD, \mathcal{Q}) \cup Ans'(v) \leftarrow Ans(v, w)$ are $Ans'(2000)$, $Ans'(2001)$, $Ans'(2002)$. □

In order to compute the consistent intervals for aggregate queries with group-by we need to create a new program Π with the following rules and facts:

(a) All the possible *Ans*-atoms from the stable models of program $\Pi(D, FD, \mathcal{Q})$, which become program facts in Π. These atoms are obtained by evaluating program $\Pi(D, FD, \mathcal{Q})$ under brave reasoning [44].

(b) All the *Ans*$'$-atoms obtained from the intersection of the stable models of repair program $\Pi(D, FD, \mathcal{Q})$ with Rule 6.10, which also become facts of program Π,

(c) The following rules:

$$glb(\bar{y}, w) \leftarrow \#min\{x : Ans(\bar{y}, x)\} = w, Ans(\bar{y}, w), Ans'(\bar{y}). \tag{6.11}$$

$$lub(\bar{y}, w) \leftarrow \#max\{x : Ans(\bar{y}, x)\} = w, Ans(\bar{y}, w), Ans'(\bar{y}). \tag{6.12}$$

Rule 6.11 computes the *greatest lower bound answer*, and Rule 6.12 computes the *least upper bound answer*, which are only computed for the groups of attributes that appear in every repair, which is achieved by adding atom $Ans'(\bar{y})$ to the rules.

Thus, the consistent answers to an aggregate query $\mathcal{Q}$ with group-by are obtained from the unique stable model of program Π.

Example 6.10 (example 6.7 and 6.9 cont.) For $\mathcal{Q}$ "`SELECT Year, max(Budget) FROM Dept GROUP BY Year`", program Π contains the following rules and facts:

$Ans'(2000)$. $Ans'(2001)$. $Ans'(2002)$. $\}$ from $\Pi(D, FD, \mathcal{Q}) \cup Ans'(v) \leftarrow Ans(v, w)$
$Ans(2000, 3000)$. $Ans(2000, 5000)$. $Ans(2001, 6000)$.
$Ans(2001, 8000)$. $Ans(2002, 7000)$.

$glb(x,w) \leftarrow \#min\{y : Ans(x,y)\} = w, Ans(x,w), Ans'(x).$

$lub(x,w) \leftarrow \#max\{y : Ans(x,y)\} = w, Ans(x,w), Ans'(x).$

The Ans'-atoms are obtained from the intersection of the stable models of program $\Pi(D, FD, \mathcal{Q}) \cup Ans'(v) \leftarrow Ans(v,w)$. The Ans-atoms are the brave answers to $\mathcal{Q}$ obtained from program $\Pi(D, FD, \mathcal{Q})$. The stable model of program Π is:

$$\begin{aligned}\mathcal{M}_1 \;=\; & \{glb(2000, 3000), glb(2001, 6000), glb(2002, 7000), lub(2000, 5000),\\ & lub(2001, 8000), lub(2002, 7000)\}.\end{aligned}$$

Hence, the consistent answers to $\mathcal{Q}$ are (2000, [3000, 5000]), (2001, [6000, 8000]) (2002, [7000]), as expected.

For query "`SELECT Name, max(Budget) FROM Dept GROUP BY Name`", the program Π has the following rules:

$Ans'(biol).\quad Ans(cs, 8000).\quad Ans(math, 6000).\quad Ans(biol, 7000).$

$glb(x,w) \leftarrow \#min\{y : Ans(x,y)\} = w, Ans(x,w), Ans'(x).$

$lub(x,w) \leftarrow \#max\{y : Ans(x,y)\} = w, Ans(x,w), Ans'(x).$

The stable model is: $\mathcal{M}_1 = \{glb(biol, 7000), lub(biol, 7000)\}$. Therefore, the consistent answer to $\mathcal{Q}$ is $(biol, 7000)$, as expected. □

Algorithm 6.2 computes consistent answers to aggregate queries of the form (6.8). The input to the algorithm consists of the database instance D, the set FD of FDs, and aggregate query $\mathcal{Q}$. It first generates the repair program $\Pi(D, FD)$, the query program $\Pi^{\mathcal{A}}(\mathcal{Q})$, and Rule 6.10, which is evaluated together with program $\Pi(D, FD, \mathcal{Q})$ under the cautious semantics to obtain the valid groups of attributes for which there exists a consistent interval. The Ans'-atoms are inserted as program facts in the new program Π.

After that, the repair program $\Pi(D, FD)$ is evaluated together with the query program $\Pi^{\mathcal{A}}(\mathcal{Q})$ under the brave semantics. In this way, every possible answers to the aggregate query from the stable models are captured. The *Ans*-atoms are also added to program Π as program facts. Finally, Rules 6.11 and 6.12 are inserted into program Π, which is evaluated in DLV system.

Algorithm 6.2: CQA-AggregateGroupBy($D, FD, \mathcal{Q}$)

```
Input: the database instance $D$, the set $FD$ of FDs, the aggregate query $\mathcal{Q}$
Output: consistent answers to query $\mathcal{Q}$
$\Pi(D, FD) := \mathit{GenerateRepairProgram}(D, FD)$;
$\Pi^{\mathcal{A}}(\mathcal{Q}) := \mathit{GenerateAProgram}(\mathcal{Q})$;
$\Pi(D, FD, \mathcal{Q}) := \Pi(D, FD) \cup \Pi^{\mathcal{A}}(\mathcal{Q})$;
$\Pi := \mathit{GenerateNewProgram}()$;
generate rule $r$ : of the form $\mathit{Ans}'(\bar{y}) \leftarrow \mathit{Ans}(\bar{y}, w)$ according to $\mathcal{Q}$;
$\Pi := \mathit{ComputeCautiousAnswers}(\Pi(D, FD) \cup \{r\})$;
$\Pi := \Pi' \cup \mathit{ComputeBraveAnswers}(\Pi(D, FD, \mathcal{Q}))$;
Add to $\Pi$ rules:
$\mathit{glb}(\bar{y}, w) \leftarrow \#\mathit{min}\{x : \mathit{Ans}(\bar{y}, x)\} = w, \mathit{Ans}(\bar{y}, w), \mathit{Ans}'(\bar{y})$.
$\mathit{lub}(\bar{y}, w) \leftarrow \#\mathit{max}\{x : \mathit{Ans}(\bar{y}, x)\} = w, \mathit{Ans}(\bar{y}, w), \mathit{Ans}'(\bar{y})$.
$\mathit{ComputeStableModel}(\Pi)$;
for each valid group $\mathit{Ans}'(\bar{y})$
  do return $((\bar{y}, [\mathit{glb}, \mathit{lub}]))$
```

6.4 Summary

In this chapter we specified logic programs to compute consistent answers to aggregate queries with both *scalar* functions, and *group-by* statements. For the former, we adopted the notion of consistent answers given in [4]. For the latter, we gave a notion of consistent answers which is based on the range semantics defined in [4] for *scalar* aggregates.

The logic programming specification with aggregate rules is restricted to FDs, and it works for the aggregate functions *max* and *min* only. Nevertheless, it should be possible to apply this approach straightforwardly to RIC-acyclic sets of ICs of the form (2.2).

We explored the aggregation capabilities of the DLV system for computing consistent answers to *scalar* aggregate queries as defined in [4]. The current version of DLV implements five aggregation functions. However, there are technical difficulties when aggregates are defined over atoms appearing in the head of disjunctive rules. In theses cases problems arise during the grounding process which is executed before the computation of the stable models. Specifically, the variable that holds the aggregate value may become unbound when the ground version of the program is computed, and DLV would have to compute all possible values for binding it. For some functions, such as *max* and *min* this problem can be solved by adding extra atoms into the aggregate rule to bind the aggregate variable, but for other functions such as *sum* that is not possible, and grounding could become more difficult.

Nevertheless, it is important to remark that this is only a technical difficulty, that should be solved in a future release of the DLV system.[3] It is important to notice that the DLV system does compute programs with aggregate functions like *sum* or *count*, but the restriction is that the aggregate atom cannot be defined by a disjunctive rule.

In [20] a method to compute normal programs with aggregates is described. Basically, normal programs with aggregates are translated into normal programs without aggregates. The stable models of the latter are used to define the semantics of the

[3]By personal communications with the authors of [36], we know that this problem is going to be fixed in a future version of DLV.

original program. The method works for stratified and unstratified non-disjunctive programs [20].

Chapter 7

Well-Founded Semantics for CQA

7.1 Introduction

The *core* of a program is the set of atoms in the intersection of all its stable models. For instance, the *core* of the original database (or of the repair program) wrt a set of ICs is the set of database atoms in the intersection of all its repairs, or equivalently, of database atoms in the intersection of all stable models of the repair program. In this chapter, we show that in some cases we can compute consistent answers to queries by using a *core computation* that can be captured by the *well-founded semantics* (WFS) of programs, as an alternative to computing and querying all the stable models. In those cases, the core of the program can be computed in polynomial time. Core computations have been considered before for CQA [4].

The *well-founded semantics* for normal logic programs was introduced in [75], and later extended to disjunctive logic programs [60, 67]. It has been used as an alternative to the stable models semantics [43, 44, 68]. In fact, if a general logic program has a total well-founded model, that model is the unique stable model [75]. Here we adopt the framework presented in [60], that defines the WFS in terms of an operator that maps interpretations to interpretations, obtaining a *well-founded interpretation* (WFI) as a least fixpoint. The WFI of a disjunctive program can be computed in polynomial time [60].

The WFI of a program is composed by three sets of atoms: the (definitely) true, the (definitely) false, and the undetermined atoms. On the other hand, the stable models semantics tries to find alternative models for the program (possibly more than one), giving to all atoms a true or false value. Therefore, a program can have several alternative stable models, but only one WFI. The stable models can be computed from the WFI by, starting from the atoms which are true or false, trying to give different values to the unknown atoms. Therefore, the set W^+ of true atoms of the WFI is contained in every stable model of the program, and the set of false atoms of the WFI is a subset of the set of atoms that are false in every stable model [60, 75].[1]

In [60] the WFI of a disjunctive program is used as a starting point to compute the stable models of a program. Moreover, in [19] the WFI of a program is used to compute the *deterministic* set of a program. This set contains the atoms that can be deterministically inferred from a program given a certain interpretation. This set is also contained in every stable model of a program.

In this chapter we explore the applicability of the WFS to CQA. We show that, under certain conditions, for UICs, RICs, and NNCs, and conjunctive queries without existential quantifiers, the core of the program $\Pi(D, IC, \mathcal{Q})$ coincides with the set of true atoms of the WFI of $\Pi(D, IC, \mathcal{Q})$. This generalizes some preliminary results obtained in [3] (for a different specification of repair programs). This property is significant, because in those cases CQA becomes polynomial in data complexity.

In addition, we take advantage of the set of undetermined atoms of the WFI, to compute consistent answers wrt functional dependencies for a restricted class of conjunctive queries with existential quantifiers. This is important because for queries containing projections (existential quantifiers), the set of true atoms of the WFI alone is not enough to retrieve all the consistent answers.

We also analyze the use of the WFS as a first step towards answering ground disjunctive queries, leaving the stable models semantics for a second stage, if necessary.

[1]In our case, stable models are sets of ground atoms, every ground atom outside this set is considered to be false.

Moreover, we consider the use of the WFS as a general way of computing consistent answers, and by doing so and by complexity theoretic reasons, just providing a lower complexity *approximation to CQA*. For example, with the WFS we retrieve a subset of the consistent answers to positive Datalog queries.

The rest of the chapter is structured as follows: in Section 7.2 the concept of core computation to CQA is presented. Section 7.3 presents the WFS of repair programs. In Section 7.4 we analyze the use of the WFS as a core computation to CQA. In particular, Section 7.5 describes the applicability of the WFS to CQA in presence of functional dependencies. Section 7.6 presents the WFS of programs as an approximation method to CQA. Section 7.7 reviews tools to compute well-founded answers. Section 7.8 summarizes this chapter.

7.2 Core Answers

The *core* of a logic program Π is the intersection of all its stable models.

Definition 7.1 For a program Π, the core of Π is:

$$\mathit{Core}(\Pi) := \bigcap\{S \mid S \text{ is a stable model of } \Pi\} \qquad \Box$$

In particular, $\mathit{Core}(\Pi(D, \mathit{IC}))$ denotes the intersection of all the stable models of the repair program $\Pi(D, \mathit{IC})$; and $\mathit{Core}(\Pi(D, \mathit{IC}, \mathcal{Q}))$ is the intersection of all the stable models of the repair program $\Pi(D, \mathit{IC})$ plus the query program $\Pi(\mathcal{Q})$. Since we are interested in the database atoms in a repair, we will restrict the core of a repair program to the atoms annotated with constant $\mathbf{t}^{\star\star}$. In this manner, the $\mathit{Core}(\Pi(D, \mathit{IC}))$ can be seen as a new database instance, which contains the database atoms that are true in every repair of D. This instance can be a repair but this may not be always the case.

Definition 7.2 Given a database instance D, a set IC of ICs, and a query $\mathcal{Q}$, a tuple of constants $\bar{t}$ is a *core answer* to $\mathcal{Q}(\bar{x})$ iff $Ans(\bar{t}) \in Core(\Pi(D, IC, \mathcal{Q}))$. If a query $\mathcal{Q}$ is an $\mathcal{L}-$sentence, i.e. a boolean query, the core answer is *yes* if $Ans \in Core(\Pi(D, IC, \mathcal{Q}))$; and *no*, otherwise. The set of core answers to a query $\mathcal{Q}$ in D wrt IC is denoted by $CoreA(\mathcal{Q})$. $\square$

Example 7.1 For database instance $D = \{S(a,b),\ S(a,c),\ S(b,c)\}$ and IC: $\forall xyz$ $(S(x,y) \wedge S(x,z) \rightarrow y = z)$. Program $\Pi(D, IC)$ has two stable models:

$$\begin{aligned}
\mathcal{M}_1 &= \{S\_(a,b,\mathbf{t}^\star), S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), S\_(a,c,\mathbf{f_a}), \underline{S\_(a,b,\mathbf{t}^{\star\star})}, \underline{S\_(b,c,\mathbf{t}^{\star\star})}\},\\
\mathcal{M}_2 &= \{S\_(a,b,\mathbf{t}^\star),\ S\_(a,c,\mathbf{t}^\star),\ S\_(b,c,\mathbf{t}^\star), S\_(a,b,\mathbf{f_a}), \underline{S\_(a,c,\mathbf{t}^{\star\star})}, \underline{S\_(b,c,\mathbf{t}^{\star\star})}\}.
\end{aligned}$$

Therefore, the database repairs are $\{S(a,b),\ S(b,c)\}$ and $\{S(a,c),\ S(b,c)\}$. Here, $Core\ (\Pi\ (D,\ IC)) = \{S\_(b,c,\mathbf{t}^{\star\star})\}$, and produces the instance $\{S(b,c)\}$, which satisfies IC, but does not minimally differ from D; hence, is not a repair of D.

For the query $Ans(x,y) \leftarrow S\_(x,y,\mathbf{t}^{\star\star})$, program $\Pi(D, FD, \mathcal{Q})$ has two stable models:

$$\begin{aligned}
\mathcal{M}_1 &= \{S\_(a,b,\mathbf{t}^{\star\star}), S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), S\_(a,c,\mathbf{f_a}), S\_(a,b,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}),\\
&\qquad \underline{Ans(a,b)}, \underline{Ans(b,c)}\},\\
\mathcal{M}_2 &= \{S\_(a,b,\mathbf{t}^{\star\star}), S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), S\_(a,b,\mathbf{f_a}), S\_(a,c,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}),\\
&\qquad \underline{Ans(a,c)}, \underline{Ans(b,c)}\}.
\end{aligned}$$

Here, $Core(\Pi(D, IC, \mathcal{Q})) = \{S\_(b,c,\mathbf{t}^{\star\star}), Ans(b,c)\}$, and hence $CoreA(\mathcal{Q}) = (b,c)$, which in this case coincides with the consistent answer to $\mathcal{Q}$. $\square$

In [4], a core computation of repairs is used to efficiently compute consistent answers to aggregate queries with scalar functions. In fact, the core of the database repairs wrt functional dependencies and the core answers are computed in polynomial time. It becomes relevant to analyze the relation between the core answers and the consistent answers to queries.

In particular, if set IC only contains *unary* ICs and NNCs of the forms (2.3) and (2.6), respectively, i.e. ICs with only one database atom in the antecedent, and a built-in in the consequent, there exists a unique database repair which coincides with the core of the repair program. Hence, for Datalog queries that, when expressed as logic programs, become normal programs, the core answers obtained from $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$ are exactly the consistent answers to queries.

Proposition 7.1 For database instance D, set IC of unary ICs and NNCs of the forms (2.3) and (2.6), respectively, and a query $\mathcal{Q}$ such that $\Pi(\mathcal{Q})$ is a Datalog normal program, the consistent answers to $\mathcal{Q}$ in D wrt IC coincide with the core answers to $\mathcal{Q}$ obtained from $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$.

Proof: Since for unary ICs (including NNCs) consistency is restored by deletion of tuples, the repair program $\Pi(D, IC)$ becomes a program without disjunction. Moreover, this program does not contain any program constraint, and therefore is locally stratified (cf. Proposition 4.3). Actually, the program $\Pi(D, IC)$ can be evaluated considering the following *strata*:

$S_0 = \{P(\bar{x}) \mid P \in \mathcal{R} \text{ and } \bar{x} \in \mathcal{U}\}$,

$S_1 = \{P\_(\bar{x}, y) \mid P \in \mathcal{R}, \bar{x} \in \mathcal{U}, \text{ and } y \in \{\mathbf{t}^\star, \mathbf{f_a}\}\}$,

$S_2 = \{P\_(\bar{x}, y) \mid P \in \mathcal{R}, \bar{x} \in \mathcal{U}, \text{ and } y \in \{\mathbf{t}^{\star\star}\}\}$

Moreover, since we are considering unary ICs, and NNCs, program $\Pi(D, IC)$ has a unique stable model S. Therefore, $\mathit{Core}(\Pi(D,\ IC))$ coincides with the model S (restricted to the atoms annotated with $\mathbf{t}^{\star\star}$). Given the fact that the query program is always a program without disjunction, and only have rules with the head predicate Ans, program $\Pi(D, IC, \mathcal{Q})$ has also a unique stable model S'. S' coincides with

$\mathit{Core}(\Pi(D, \mathit{IC}, \mathcal{Q}))$ (restricted to the atoms annotated with $\mathbf{t}^{\star\star}$ plus the Ans-atoms). It follows that a tuple $\bar{t}$ is a consistent answer to $\mathcal{Q}$ in D wrt IC iff $\mathit{Ans}(\bar{t})$ is in $\mathit{Core}(\Pi(D, \mathit{IC}, \mathcal{Q}))$. $\Box$

Example 7.2 For database instance $D = \{\mathit{Emp}(\mathit{john},\ \mathit{ceo},\ 10),\ \mathit{Emp}(\mathit{mary},\ \mathit{eng},\ 20),\ \mathit{Emp}(\mathit{peter},\ \mathit{ceo},\ 30)\}$ and IC: $\forall xyz(\mathit{Emp}(x,y,z) \wedge y = \mathit{ceo} \rightarrow z > 20)$. Program $\Pi(D, \mathit{IC})$ has one stable model:

$$\begin{aligned}\mathcal{M}_1 \;=\; & \{\mathit{Emp\_}(\mathit{john}, \mathit{ceo}, 10, \mathbf{f_a}), \mathit{Emp\_}(\mathit{john}, \mathit{ceo}, 10, \mathbf{t}^\star), \mathit{Emp\_}(\mathit{mary}, \mathit{eng}, 20, \mathbf{t}^\star),\\ & \quad \mathit{Emp\_}(\mathit{peter}, \mathit{ceo}, 30, \mathbf{t}^\star), \underline{\mathit{Emp\_}(\mathit{mary}, \mathit{eng}, 20, \mathbf{t}^{\star\star})}, \underline{\mathit{Emp\_}(\mathit{peter}, \mathit{ceo}, 30, \mathbf{t}^{\star\star})}\}.\end{aligned}$$

Thus, $\mathit{Core}(\Pi(D, \mathit{IC})) = \{\mathit{Emp}(\mathit{mary},\ \mathit{eng},\ 20, \mathbf{t}^{\star\star}), \mathit{Emp}(\mathit{peter},\ \mathit{ceo},\ 30, \mathbf{t}^{\star\star})\}$, producing the instance $\{\mathit{Emp}(\mathit{mary},\ \mathit{eng},\ 20),\ \mathit{Emp}(\mathit{peter},\ \mathit{ceo},\ 30)\}$, which corresponds to the unique database repair for D.

For query $\mathit{Ans}(x) \leftarrow \mathit{Emp\_}(x, y,\ z, \mathbf{t}^{\star\star})$, $\mathit{Core}(\Pi(D, \mathit{IC}, \mathcal{Q})) = \{\mathit{Emp}(\mathit{mary},\ \mathit{eng},\ 20, \mathbf{t}^{\star\star}), \mathit{Emp}(\mathit{peter},\ \mathit{ceo},\ 30, \mathbf{t}^{\star\star}), \mathit{Ans}(\mathit{peter}),\ \mathit{Ans}(\mathit{mary})\}$. Therefore, the core answers are $(\mathit{peter}), (\mathit{mary})$, which are also the consistent answers to $\mathcal{Q}$. $\Box$

In the general case of RIC-acyclic ICs, and for conjunctive queries without existential quantifiers, the core answers to queries, $\mathit{CoreA}(\mathcal{Q})$, coincide with their consistent answers, $\mathit{ConsA}(\mathcal{Q})$.[2]

Proposition 7.2 For a RIC-acyclic set IC of UICs, RICs and NNCs of the forms (2.3), (2.4), and (2.6), respectively, and a conjunctive query $\mathcal{Q}$ without existential quantifiers, the consistent answers to $\mathcal{Q}$ in D wrt IC, $\mathit{ConsA}(\mathcal{Q})$, coincide with the core answers to $\mathcal{Q}$, $\mathit{CoreA}(\mathcal{Q})$, obtained from $\mathit{Core}(\Pi(D, \mathit{IC}, \mathcal{Q}))$.

[2]As in Definition 2.12, $\mathit{ConsA}(\mathcal{Q})$ denotes the set of consistent answers to a query $\mathcal{Q}$.

Proof: We need to prove that if a tuple $\bar{t}$ is a consistent answer to a query $\mathcal{Q}$, then $Ans(\bar{t})$ is in $Core(\Pi(D, IC, \mathcal{Q}))$, and viceversa. First, we prove that if a tuple $\bar{a}$ is a consistent answer to $\mathcal{Q}$, then $Ans(\bar{a})$ is in $Core(\Pi(D, IC, \mathcal{Q}))$. Second, we show that if an atom $Ans(\bar{a})$ is in $Core(\Pi(D, IC, \mathcal{Q}))$ then the tuple $\bar{a}$ is a consistent answer to $\mathcal{Q}$.

(a) By contradiction, let us assume that there exists a tuple $\bar{a}$ that is a consistent answer to $\mathcal{Q}$, but $Ans(\bar{a})$ is not in $Core(\Pi(D, IC, \mathcal{Q}))$. If this happen, then all the query atoms of the form $P\_(\bar{c}, \mathbf{t}^{\star\star})$ on $\mathcal{Q}$ with $\bar{a} \subseteq \bar{c}$ defining $Ans(\bar{a})$ are true in every repair. Since the query atoms are true in every repair, they appear in $Core(\Pi(D, IC))$. As a consequence, $Ans(\bar{a})$ is in $Core(\Pi(D, IC, \mathcal{Q}))$. We have reached a contradiction.

(b) By contradiction, let us assume that there exists an atom $Ans(\bar{a})$ that is in $Core(\Pi(D, IC, \mathcal{Q}))$ but tuple $\bar{a}$ is not a consistent answer to $\mathcal{Q}$. Since the query program does not contain rules defining any of the query predicates, and since $Ans(\bar{a})$ is in $Core(\Pi(D, IC, \mathcal{Q}))$, then all the query atoms of the form $P\_(\bar{c}, \mathbf{t}^{\star\star})$ on $\mathcal{Q}$ with $\bar{a} \subseteq \bar{c}$ are true in every repair, because the query does not have existentially quantified variables. Therefore, tuple $\bar{a}$ has to be a consistent answer to $\mathcal{Q}$. We have reached a contradiction. $\square$

This result is important because for conjunctive queries without existential quantifiers, we can compute consistent answers by focusing on the core of the program $\Pi(D, IC, \mathcal{Q})$, avoiding the full computation of all the stable models. This is assuming that the core of a program can be obtained without the generation of every stable model of the program. For queries with existential quantifiers, it would be possible that the core of program $\Pi(D, IC, \mathcal{Q})$ will not retrieve all the consistent answers. We will illustrate this in Example 7.3.

In this chapter, we show that the well-founded interpretation of programs can be considered as a core computation to CQA. As a matter of fact, in Section 7.4 we prove that for a RIC-acyclic set of ICs, that is *interaction-free* (cf. Definition 7.6), the core of a repair coincides with the set of true atoms of the WFI of program $\Pi(D, IC)$ (cf. Theorem 7.1). Intuitively, a set of ICs is *interaction-free* if there is no interaction between unary ICs (and NNCs) and the other ICs, or between RICs and other ICs. Moreover, for conjunctive queries without existential quantifiers it also holds that the core of program $\Pi(D, IC, \mathcal{Q})$ coincides with set of true atoms of the WFI of program $\Pi(D, IC, \mathcal{Q})$ (cf. Theorem 7.1).

In this manner, the core of a program can be obtained without computing all its stable models. Moreover, it can be computed in polynomial time, and as a consequence CQA for conjunctive queries without projections wrt *interaction-free* sets of RIC-acyclic ICs becomes polynomial (cf. Corollary 7.1).

In [2, 22, 27, 39] there are polynomial time algorithms for consistently answering this kind of queries wrt UICs. The methods in [27, 39] also apply to conjunctive queries with restricted forms of projection, obtaining for them also polynomial time.

However, we cannot always compute consistent answers to existentially quantified conjunctive queries, or disjunctive queries by using a core computation alone. This is illustrated in the example below.

Example 7.3 (example 7.1 cont.) For the existential query: $Ans(y) \leftarrow S\_(x, y, \mathbf{t}^{\star\star})$,[3] $CoreA(\mathcal{Q})= (c)$, that coincides with the consistent answer to $\mathcal{Q}$. For the ground disjunctive query $Ans \leftarrow S\_(b, c, \mathbf{t}^{\star\star}) \vee S\_(a, b, \mathbf{t}^{\star\star})$, which program contains rules: $Ans \leftarrow S\_(b, c, \mathbf{t}^{\star\star})$, and $Ans \leftarrow S\_(a, b, \mathbf{t}^{\star\star})$, both the core and the consistent answers are *yes*. For the boolean query $Ans \leftarrow S(x_1, y), S(x_2, y),\ x_1 \neq x_2$, where the existentially quantifier variables are x_1, x_2, the core and the consistent answers are *no*.

[3]In this query, the variable x is existentially quantified.

However, for $\mathcal{Q}$: $Ans(x) \leftarrow S\_(x, y, \mathbf{t}^{\star\star})$, $CoreA(\mathcal{Q})$= (b), but the consistent answers are $(b), (a)$. For $\mathcal{Q}$: $Ans \leftarrow S\_(a, b, \mathbf{t}^{\star\star}) \vee S\_(a, c, \mathbf{t}^{\star\star})$, the query program contains rules: $Ans \leftarrow S\_(a, b, \mathbf{t}^{\star\star})$, and $Ans \leftarrow S\_(a, c, \mathbf{t}^{\star\star})$, the consistent answer is *yes*, but the core answer is *no*. This happens because both queries involve tuples that appear in different repairs, which are not captured in $Core(\Pi(D, IC, \mathcal{Q}))$. $\Box$

Nevertheless, we can use a core computation as a first step to retrieve query answers, leaving the stable model computation as a possible second stage. This idea is developed in Section 7.6. In the following section, we explore the use of the well-founded semantics of programs as a core computation to CQA.

7.3 Well-Founded Semantics of Repair Programs

The *well-founded interpretation* for a ground disjunctive program Π consists of three disjoint and complementary sets of ground atoms: $W_\Pi = \langle W^+,\ W^-,\ W^u\rangle$, where W^+ is the set of true atoms, W^- is the set of false atoms, and W^u is the set of undetermined atoms [60]. The WFI is defined as the least fixpoint of an operator $\mathcal{W}_\Pi$, that is a mapping between interpretations of the form $I = \langle I^+, I^-, I^u\rangle$, with I^+, I^-, I^u disjoint sets of ground atoms that cover the whole Herbrand base of the program.

If we define a *literal* as a formula of the form A or *not* A, with A atomic, then interpretations I can be represented as sets of ground literals. In this case, I^+ is the set of atoms (i.e. positive literals) in I, and I^- is $not.\{\ not\ A \mid\ not\ A \in I\}$,[4] and $I^u = \{A \mid A$ is ground atom and both $A,\ not\ A \notin I\}$. On the other hand, an interpretation of the form $I = \langle I^+,\ I^-,\ I^u\rangle$ can be represented as the set of literals $I = I^+ \cup\ not.I^-$, where $not.I^- = \{\ not\ A \mid\ A$ is an atom in $I^-\}$. Then, I^u

[4]Remark $not.\mathcal{L}$, with $\mathcal{L}$ a set of literals, is the set of literals that are complementary to those in $\mathcal{L}$. For a literal L, $not.L$ denotes the literal that is complementary to L.

becomes implicitly the set of atoms A such that neither A nor *not* A can be found in $I^+ \cup$ $not.I^-$.

From now on, an interpretation will be a set I of ground literals, that is I is a subset of $B_\Pi \cup not.B_\Pi$,[5] such that for no atom A, both A and *not* A belong to I. For a ground literal L, L is true (false) wrt I if $L \in I$ ($not.L \in I$). A literal L is undetermined in I if it is neither true nor false in I.

The operator $\mathcal{W}_\Pi$ is based in an extension of the notion of *unfounded set* to disjunctive programs (Definition 7.3). An unfounded set is a subset of atoms from the Herbrand base B_Π of program Π. Unfounded sets single out the atoms that are definitely not derivable from a given program wrt a given interpretation, and as a consequence, they are declared false.

Definition 7.3 [60] Let I be an interpretation for (the ground version of a) program Π. A set $X \subseteq B_\Pi$ of ground atoms is an *unfounded set* for Π wrt I if for each $a \in X$ (an unfounded atom in X), for each rule $r \in \Pi$ (the instantiated i.e. ground version of Π), such that $a \in H(r)$, the head of rule r, at least one of the following conditions holds:

(a) $B(r) \cap$ $not.I \neq \emptyset$, i.e. the body of r is false regarding I.

(b) $B^+(r) \cap X \neq \emptyset$, i.e. some positive body literal belongs to X.

(c) $(H(r) \smallsetminus X) \cap I \neq \emptyset$, i.e. an atom in the head of r, distinct from a and other elements in X, is true wrt I. $\square$

The union of all the unfounded sets for a program Π regarding an interpretation I, $GUS_\Pi(I)$, is called the *greatest unfounded set wrt* I. For normal programs, the GUS is also an unfounded set, but for disjunctive programs this might not be the case

[5]As a reminder, B_Π stands for the Herbrand base of a program Π.

(so we might have a greatest unfounded set that is not unfounded) [60]. However, $GUS_{\Pi}(I)$ is unfounded when I is an *unfounded-free interpretation*, i.e. when it has empty intersection with every set that is unfounded regarding I [60].

Definition 7.4 Given a ground disjunctive program Π, the *well-founded operator* (WFO), denoted by $\mathcal{W}_{\Pi}$, is defined on interpretations I for which $GUS_{\Pi}(I)$ is unfounded, by:

$$\mathcal{W}_{\Pi}(I) := \Gamma_{\Pi}(I) \ \cup \ not.GUS_{\Pi}(I),$$

where $\Gamma_{\Pi}(I)$ is the immediate consequence operator that declares an atom A true wrt I if there exists a rule in Π, such that A is in the head of the rule, the body of the rule is true wrt I, and the other atoms in the head of the rule (if any) are false wrt I. □

Definition 7.5 The *well founded interpretation* (WFI) of a ground program Π is defined as the fixpoint W_{Π} of the interpretations defined by:

$$W_0 := \emptyset,$$

$$W_{k+1} := \mathcal{W}_{\Pi}(W_k).$$

W_{Π} can be computed in polynomial time [60]. □

Moreover, we can guarantee that for our repair programs, GUS is always unfounded. This given the following result proved in [60].

Proposition 7.3 [60] For a program Π, an interpretation I, and a stable model M of Π. If $I \subseteq M$, then (a) I is *unfounded-free.* (b) $W_{\Pi} \subseteq M$. □

Therefore, since the repair programs always have stable models (it is guarantee that every database has a repair [2]), we have that $I = \emptyset$ is *unfounded-free*, hence the $GUS_{\Pi}(I)$ is always unfounded.

Moreover, in [60] it is proven that for every function-free program Π, the W_{Π} as defined in Definition 7.5 always reaches a fixpoint.

Example 7.4 (example 7.1 cont.) For database instance $D = \{S(a,b),\ S(a,c),\ S(b,c)\}$, and IC: $\forall xyz(S(x,y) \wedge S(x,z) \rightarrow y = z)$, program $\Pi(D, IC)$ contains the following rules:

$S(a,b).\quad S(a,c).\quad S(b,c).$

$S\_(x,y,\mathbf{f_a}) \vee S\_(x,z,\mathbf{f_a}) \leftarrow S\_(x,y,\mathbf{t^\star}), S\_(x,z,\mathbf{t^\star}), y \neq z, x \neq null, y \neq null, z \neq null.$

$S\_(x,y,\mathbf{t^\star}) \leftarrow S(x,y).$

$S\_(x,y,\mathbf{t^\star}) \leftarrow S\_(x,y,\mathbf{t_a}).$

$S\_(x,y,\mathbf{t^{\star\star}}) \leftarrow S\_(x,y,\mathbf{t^\star}),\ not\ S\_(x,y,\mathbf{f_a}).$

$\mathcal{W}_{\Pi}$ is defined as follows:

1. For $W_0 = \emptyset$ the unfounded set is:

$$GUS_{\Pi}(W_0) = \{S(a,a), S(b,b), S(c,c), S(b,a), S(c,a), S(c,b)\},$$

   which is composed by the atoms that are false regarding I. Specifically, these atoms do not appear in the head of any instantiated rule, therefore they are immediately put into the unfounded set, because the conditions for unfoundedness are checked for atoms in heads.

   Thus, $\mathcal{W}_{\Pi}(W_0) = \{S(a,b), S(a,c), S(b,c)\} \cup not.GUS_{\Pi}(W_0)$, i.e. the database atoms in the program, plus the complement of the unfounded set constructed so far.

2. For $W_1 = \mathcal{W}_\Pi(W_0)$, $GUS_\Pi(W_1)$ is composed by the set of atoms in $GUS_\Pi(W_0)$ plus the atoms that are unfounded wrt $\mathcal{W}_\Pi(W_0)$.

   Here, all the atoms of the form $S\_(\bar{c}, \mathbf{t_a})$ are declare unfounded, because there are no rules defining them in the program. Also, the atoms defined by annotation rules of the form $S\_(\bar{c}, \mathbf{t}^\star) \leftarrow S\_(\bar{c}, \mathbf{t_a})$, for which the body becomes false, are also declared unfounded (cf. item (a) in Definition 7.3).

   Therefore, the greatest unfounded set is:

$$\begin{aligned} GUS_\Pi(W_1) \;\; = \;\; & GUS_\Pi(W_0) \cup \{S\_(a,a,\mathbf{t}^\star), S\_(b,b,\mathbf{t}^\star), S\_(c,c,\mathbf{t}^\star), S\_(b,a,\mathbf{t}^\star),\\ & S\_(c,a,\mathbf{t}^\star), S\_(c,b,\mathbf{t}^\star), S\_(a,a,\mathbf{t_a}), S\_(b,b,\mathbf{t_a}), S\_(c,c,\mathbf{t_a}),\\ & S\_(b,a,\mathbf{t_a}), S\_(c,a,\mathbf{t_a}), S\_(c,b,\mathbf{t_a}), S\_(a,b,\mathbf{t_a}), S\_(a,c,\mathbf{t_a}),\\ & S\_(b,c,\mathbf{t_a})\}. \end{aligned}$$

   Now, since atoms $S(a,b)$, $S(a,c)$, $S(b,c)$ are in $\mathcal{W}_\Pi(W_0)$, we obtain atoms $S\_(a,b,\mathbf{t}^\star)$, $S\_(a,c,\mathbf{t}^\star)$, $S\_(b,c,\mathbf{t}^\star)$, by using the annotation rules of the form $S\_(\bar{c}, \mathbf{t}^\star) \leftarrow S(\bar{c})$. Hence,

$$\begin{aligned} \mathcal{W}_\Pi(W_1) \;\; = \;\; & \{S(a,b), S(a,c), S(b,c), S\_(a,b,\mathbf{t}^\star), S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star)\}\\ & \cup \; not.GUS_\Pi(W_1). \end{aligned}$$

3. For $W_2 = \mathcal{W}_\Pi(W_1)$, $GUS_\Pi(W_2)$ is composed by the set of atoms in $GUS_\Pi(W_1)$ plus the atoms that are unfounded wrt $\mathcal{W}_\Pi(W_1)$.

   Here, all the atoms of the form $S\_(\bar{c}, \mathbf{f_a})$, for which the body becomes false, are declared unfounded. There are no new positive consequences of the program. Hence,

$$\begin{aligned} GUS_\Pi(W_2) &= GUS_\Pi(W_1) \cup \{S\_(b,c,\mathbf{f_a}), S\_(b,a,\mathbf{f_a}), S\_(b,b,\mathbf{f_a}), S\_(a,a,\mathbf{f_a}), \\ &\quad S\_(c,c,\mathbf{f_a}), S\_(c,a,\mathbf{f_a}), S\_(c,b,\mathbf{f_a})\}. \\ \mathcal{W}_\Pi(W_2) &= \{S(a,b), S(a,c), S(b,c), S\_(a,b,\mathbf{t}^\star), S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star)\} \\ &\quad \cup\ not.GUS_\Pi(W_2). \end{aligned}$$

4. For $W_3 = \mathcal{W}_\Pi(W_2)$, $GUS_\Pi(W_3)$ is composed by the set of atoms in $GUS_\Pi(W_2)$ plus the atoms that are unfounded wrt $\mathcal{W}_\Pi(W_2)$.

   Here, all the atoms of the form $S\_(\bar{c}, \mathbf{t}^{\star\star})$, for which the body becomes false are declared unfounded. Atom $S\_(b,c,\mathbf{t}^{\star\star})$ is a positive consequence of the program. Therefore,

$$\begin{aligned} GUS_\Pi(W_3) &= GUS_\Pi(W_2) \cup \{S\_(a,a,\mathbf{t}^{\star\star}), S\_(b,b,\mathbf{t}^{\star\star}), S\_(c,c,\mathbf{t}^{\star\star}), S\_(b,a,\mathbf{t}^{\star\star}), \\ &\quad S\_(c,a,\mathbf{t}^{\star\star}), S\_(c,b,\mathbf{t}^{\star\star})\}. \\ \mathcal{W}_\Pi(W_3) &= \{S(a,b), S(a,c), S(b,c), S\_(a,b,\mathbf{t}^\star), S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), \\ &\quad S\_(b,c,\mathbf{t}^{\star\star})\} \cup\ not.GUS_\Pi(W_3). \end{aligned}$$

5. $W_4 = \mathcal{W}_\Pi(W_3) = \mathcal{W}_\Pi(W_4)$. We have reached a fixpoint.

The WFI of the program $\Pi(D, IC)$ is:

$$\begin{aligned} W^+ &= \{S(a,b), S(a,c), S(b,c), S\_(a,b,\mathbf{t}^\star), S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^{\star\star})\}, \\ W^u &= \{S\_(a,b,\mathbf{f_a}), S\_(a,b,\mathbf{t}^{\star\star}), S\_(a,c,\mathbf{f_a}), S\_(a,c,\mathbf{t}^{\star\star})\}, \end{aligned}$$

$$\begin{aligned} W^- = \{ & S(a,a), S(b,b), S(c,c), S(b,a), S(c,a), S(c,b), S\_(a,a,\mathbf{t^\star}), S\_(b,b,\mathbf{t^\star}), \\ & S\_(c,c,\mathbf{t^\star}), S\_(b,a,\mathbf{t^\star}), S\_(c,a,\mathbf{t^\star}), S\_(c,b,\mathbf{t^\star}), S\_(a,a,\mathbf{t_a}), S\_(b,b,\mathbf{t_a}), \\ & S\_(c,c,\mathbf{t_a}), S\_(b,a,\mathbf{t_a}), S\_(c,a,\mathbf{t_a}), S\_(c,b,\mathbf{t_a}), S\_(a,b,\mathbf{t_a}), S\_(a,c,\mathbf{t_a}), \\ & S\_(b,c,\mathbf{t_a}), \{S\_(b,c,\mathbf{f_a}), S\_(b,a,\mathbf{f_a}), S\_(b,b,\mathbf{f_a}), S\_(a,a,\mathbf{f_a}), S\_(c,c,\mathbf{f_a}), \\ & S\_(c,a,\mathbf{f_a}), S\_(c,b,\mathbf{f_a}), S\_(a,a,\mathbf{t^{\star\star}}), S\_(b,b,\mathbf{t^{\star\star}}), S\_(c,c,\mathbf{t^{\star\star}}), S\_(b,a,\mathbf{t^{\star\star}}), \\ & S\_(c,a,\mathbf{t^{\star\star}}), S\_(c,b,\mathbf{t^{\star\star}})\}. \end{aligned}$$

From the set W^+ of the positive atoms of the WFI of $\Pi(D, IC)$, we obtain the database instance $D' = \{S(b,c)\}$ ($S\_(b,c,\mathbf{t^{\star\star}}) \in W^+$). Here D' is consistent wrt IC, but not a repair of D. Actually, the repairs are: $\{S(b,c), S(a,b)\}$ and $\{S(b,c), S(a,c)\}$. But atoms $S\_(a,b,\mathbf{t^{\star\star}})$, $S\_(a,c,\mathbf{t^{\star\star}})$ are undetermined in the WFI of $\Pi(D, IC)$. Actually, since there are undetermined atoms, the WFI of the repair program $\Pi(D, IC)$ cannot be considered as a model of $\Pi(D, IC)$. In fact, it does not capture the semantics of the program, i.e. the database repairs. □

It is important to notice that for a disjunctive repair program, the WFI is usually not a model of the program, but only an interpretation [60].[6] This happens because the WFI does not conclude anything positive from disjunctive rules, and therefore head atoms in these rules are declared as undetermined. As an illustration, in Example 7.4, atoms $S\_(a,b,\mathbf{f_a}), S\_(a,c,\mathbf{f_a})$, which are in the head of the only disjunctive rule in $\Pi(D, IC)$, are declared as undetermined, and as a consequence the atoms $S\_(a,b,\mathbf{t^{\star\star}}), S\_(a,c,\mathbf{t^{\star\star}})$ become undetermined.

In our setting, stable models are sets of ground atoms. A stable model is a total interpretations (and a model), because any ground atom that is not in it is considered to be false, by applying a closed world assumption to the model [69]. If

[6]A model is a total interpretation I, with $I^u = \emptyset$, that makes the program true.

a stable model $\mathcal{M}$ is conceived as an interpretation for the program, we would have $\mathcal{M}^+ = \mathcal{M}, \mathcal{M}^- = \{A \mid A \text{ is ground atom and } A \notin \mathcal{M}\}$, $\mathcal{M}^u = \emptyset$. The true and false atoms in the WFI of a program Π are contained in the intersection of all the stable model of Π, in the sense that $W^+ \subseteq \bigcap\{\mathcal{M} \mid \mathcal{M} \text{ is a stable model of } \Pi\}$, and $W^- \subseteq \bigcap\{\mathcal{M}^- \mid \mathcal{M} \text{ is a stable model of } \Pi\}$ [60].

Example 7.5 (example 7.4 cont.) Program $\Pi(D, IC)$ has two stable models:

$$\begin{aligned}\mathcal{M}_1 &= \{S\_(a,b,\mathbf{t^\star}), S\_(a,c,\mathbf{t^\star}), S\_(b,c,\mathbf{t^\star}), S\_(a,c,\mathbf{f_a}), S\_(a,b,\mathbf{t^{\star\star}}), S\_(b,c,\mathbf{t^{\star\star}})\},\\ \mathcal{M}_2 &= \{S\_(a,b,\mathbf{t^\star}), S\_(a,c,\mathbf{t^\star}), S\_(b,c,\mathbf{t^\star}), S\_(a,b,\mathbf{f_a}), S\_(a,c,\mathbf{t^{\star\star}}), S\_(b,c,\mathbf{t^{\star\star}})\}.\end{aligned}$$

Therefore, $\mathcal{M}_1^+ = \mathcal{M}_1$, $\mathcal{M}_1^- = \{S\_(a,a,\mathbf{t^\star}),\ S\_(b,b,\mathbf{t^\star}),\ S\_(a,c,\mathbf{t^{\star\star}}),\ \ldots\}$; $\mathcal{M}_2^+ = \mathcal{M}_2$, $\mathcal{M}_2^- = \{S\_(a,a,\mathbf{t^\star}),\ S\_(b,b,\mathbf{t^\star}),\ S\_(a,b,\mathbf{t^{\star\star}}),\ \ldots\}$. The set of true atoms of the WFI of the program (without considering program facts) is:

$$W^+ = \{S\_(a,b,\mathbf{t^\star}), S\_(a,c,\mathbf{t^\star}), S\_(b,c,\mathbf{t^\star}), S\_(b,c,\mathbf{t^{\star\star}})\},$$

and $W^+ \subseteq \mathcal{M}_1^+ \bigcap \mathcal{M}_2^+$, actually as a matter of fact, they coincide. The set of false atoms is:

$$W^- = \{S\_(a,a,\mathbf{t^\star}), S\_(b,b,\mathbf{t^\star}), S\_(c,c,\mathbf{t^\star}), S\_(b,a,\mathbf{t^\star}), S\_(c,a,\mathbf{t^\star}), S\_(c,b,\mathbf{t^\star}), \ldots\},$$

and $W^- \subseteq \mathcal{M}_1^- \bigcap \mathcal{M}_2^-$. The atoms that are declared as undetermined in the WFI of the program, i.e. set $W^u = \{S\_(a,b,\mathbf{f_a}),\ S\_(a,b,\mathbf{t^{\star\star}}),\ S\_(a,c,\mathbf{f_a}),\ S\_(a,c,\mathbf{t^{\star\star}})\}$, are not undetermined in the stable models, but we have $\{S\_(a,c,\mathbf{f_a}),\ S\_(a,b,\mathbf{t^{\star\star}})\} \subseteq \mathcal{M}_1$, and $\{S\_(a,b,\mathbf{f_a}),\ S\_(a,c,\mathbf{t^{\star\star}})\} \subseteq \mathcal{M}_2$. $\square$

7.4 The Well-Founded Semantics as a Core Computation to CQA

Even though the WFI of a program may not capture the semantics of the program, it can be used as a core computation to CQA. In general, the WFI of a disjunctive program is contained in the intersection of all its stable models [60]. However, we will show that, for a restricted set of ICs and queries, the set W^+ (restricted to the *Ans*-atoms) of the WFI of program $\Pi(D, IC, \mathcal{Q})$ coincides with the core of program $\Pi(D, IC, \mathcal{Q})$.

Definition 7.6 A RIC-acyclic set *IC* of UICs, RICs and NNCs of the forms (2.3), (2.4), and (2.6), respectively, is *interaction-free* if the following holds: (a) If there is a unary IC or NNC on relation P, there is no other IC in *IC* having P in its consequent. (b) If there exists a RIC with P in its consequent, there is no other IC in *IC* having P neither in its antecedent nor its consequent. $\Box$

In other words, a RIC-acyclic set of ICs is *interaction-free* if for a given set of ICs there is no interaction between unary ICs (and NNCs) and other ICs, or between RICs and other ICs.

We now show that for *interaction-free* sets of ICs, and conjunctive queries without projections, the WFI of programs provides the same consistent answers as the stable model semantics. Notice that we are considering generic ICs [10], in the sense that they do not enforce the presence of any atom in the database.

Theorem 7.1 For a database instance D, an *interaction-free* set *IC* of ICs, and conjunctive queries without existential quantifiers, the following holds: (a) $\mathit{Core}(\Pi(D, IC))$ coincides with set W^+ of the WFI of program $\Pi(D, IC)$, restricted both to the atoms annotated with constant $\mathbf{t}^{\star\star}$. (b) $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$ coincides with W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$, restricted both to the *Ans*-atoms.

Proof: The proof is divided in two parts. First we show that for a given database instance D, and an *interaction-free* set IC of ICs, $\mathit{Core}(\Pi(D, IC))$ and W^+ of the WFI of program $\Pi(D, IC)$ coincide. Second, we prove that for conjunctive queries without projection, $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$ coincides with W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$.

(a) Since it is always the case that the WFI of a program Π is contained in the core of the program [60], we only need to show that $\mathit{Core}(\Pi(D, IC)) \subseteqq W_{\Pi(D,IC)}$.

It is necessary to show that, whenever an atom of the form $P(\bar{a})$, $P\_(\bar{a}, y)$, $aux(\bar{a})$ belongs to $\mathit{Core}(\Pi(D, IC))$, where $\bar{a}$ is a tuple of elements in the database domain $\mathcal{U}$, and $y \in \{\mathbf{f_a}, \mathbf{t_a}, \mathbf{t}^\star, \mathbf{t}^{\star\star}\}$, it holds that $P(\bar{a})$ (resp. $P\_(\bar{a}, y)$, $aux(\bar{a})$) can be fetched into $W^n_\Pi(\emptyset)$ for some integer n.

We need to analyze the possible cases for the atom:

1. $P(\bar{a}) \in \mathit{Core}(\Pi(D, IC))$. To prove: $P(\bar{a}) \in W_{\Pi(D,IC)}$.

   $P(\bar{a})$ is database fact, then $P(\bar{a})$ is fetched into $W^1_\Pi(\emptyset)$.

2. $P\_(\bar{a}, \mathbf{t_a}) \in \mathit{Core}(\Pi(D, IC))$. To prove: $P\_(\bar{a}, \mathbf{t_a}) \in W_{\Pi(D,IC)}$.

   By contradiction, if an atom of the form $P\_(\bar{a}, \mathbf{t_a})$ is true in the program, then there exists a disjunctive rule r with $P\_(\bar{a}, \mathbf{t_a})$ in $H(r)$, and whose body is true in the program. However, if this happens, and given the fact that the set of ICs is *interaction-free*, some models will get $P\_(\bar{a}, \mathbf{t_a})$, and other models will get other head atoms in $H(r)$, and as a consequence atom $P\_(\bar{a}, \mathbf{t_a})$ cannot be in $\mathit{Core}(\Pi(D, IC))$. But, $P\_(\bar{a}, \mathbf{t_a})$ is in the core. We have reached a contradiction.

3. $P\_(\bar{a}, \mathbf{t}^\star) \in \mathit{Core}(\Pi(D, IC))$. To prove: $P\_(\bar{a}, \mathbf{t}^\star) \in W_{\Pi(D,IC)}$.

   If $P\_(\bar{a}, \mathbf{t}^\star)$ belongs to the core, then there are two cases: first, $P\_(\bar{a}, \mathbf{t}^\star)$ is in the core because $P(\bar{a})$ is a program fact, and as a consequence rule

$P\_(\bar{a}, \mathbf{t^\star}) \leftarrow P(\bar{a})$ is satisfied in the program. In this case, due to the fact that $P(\bar{a})$ is in $W^1_\Pi(\emptyset)$, then $P\_(\bar{a}, \mathbf{t^\star})$ is in $W^2_\Pi(\emptyset)$.

The second case is when $P\_(\bar{a}, \mathbf{t^\star})$ is true in the core because atom $P\_(\bar{a}, \mathbf{t_a})$ is true in the program, and as a consequence rule $P\_(\bar{a}, \mathbf{t^\star}) \leftarrow P\_(\bar{a}, \mathbf{t_a})$ is satisfied in the program. However, if $P\_(\bar{a}, \mathbf{t_a})$ is true in the program, then atom $P\_(\bar{a}, \mathbf{t^\star})$ cannot be true in the core. This is because, as we showed in item 2, in this case, some models will have $P\_(\bar{a}, \mathbf{t_a})$ and some will not. Then, it is easy to see that some models will have $P\_(\bar{a}, \mathbf{t^\star})$ and others will not. Therefore, the only way to have $P\_(\bar{a}, \mathbf{t^\star})$ in the core, is because $P(\bar{a})$ is a program fact, and in this case it also belongs to the well-founded interpretation.

4. $P\_(\bar{a}, \mathbf{f_a}) \in \mathit{Core}(\Pi(D, IC))$. To prove: $P\_(\bar{a}, \mathbf{f_a}) \in W_{\Pi(D,IC)}$.

   If $P\_(\bar{a}, \mathbf{f_a})$ is in the core, then atom $P(\bar{a})$ participates in a violation of a IC. There are two cases to evaluate. First, the IC is a unary IC or a NNC. Let us assume that it is unary IC of the form: $P(\bar{a}) \rightarrow \varphi(\bar{a})$, with $P(\bar{a})$ true in the program and $\varphi(\bar{a})$ false. Therefore, in order to generate atom $P\_(\bar{a}, \mathbf{f_a})$, the following rule is satisfied, with a true body, in the program: $P\_(\bar{a}, \mathbf{f_a}) \leftarrow P\_(\bar{a}, \mathbf{t^\star}), \bar{\varphi}(\bar{a}), \bar{a} \neq \mathit{null}$, where $\bar{\varphi}$ is equivalent to the negation of φ. Given the fact that $P\_(\bar{a}, \mathbf{t^\star})$ is in $W^2_\Pi(\emptyset)$, and the built-in $\bar{\varphi}(\bar{a})$ is true in the program, then $P\_(\bar{a}, \mathbf{f_a})$ is in $W^3_\Pi(\emptyset)$.

   The second case is when atom $P(\bar{a})$ is involved in a violation of an IC having more than one database atom. We prove that in this case it is not possible to have $P\_(\bar{a}, \mathbf{f_a})$ in the core. If an atom of the form $P\_(\bar{a}, \mathbf{f_a})$ is true in the program, then there is a disjunctive rule r with $P\_(\bar{a}, \mathbf{f_a})$ in $H(r)$. However, if this happens, and given the fact that the set of ICs is *interaction-free*, some models will get $P\_(\bar{a}, \mathbf{f_a})$ and other models will get other atoms in $H(r)$, and as a consequence atom $P\_(\bar{a}, \mathbf{f_a})$ cannot be in $Core(\Pi(D, IC))$.

5. $P\_(\bar{a}, \mathbf{t}^{\star\star}) \in Core(\Pi(D, IC))$. To prove: $P\_(\bar{a}, \mathbf{t}^{\star\star}) \in W_{\Pi(D,IC)}$.

   If $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is true in the core, then $P\_(\bar{a}, \mathbf{t}^{\star})$ is true in the program and $P\_(\bar{a}, \mathbf{f_a})$ is false; then rule $P\_(\bar{a}, \mathbf{t}^{\star\star}) \leftarrow P\_(\bar{a}, \mathbf{t}^{\star})$, not $P\_(\bar{a}, \mathbf{f_a})$, is satisfied in the program. There is only one possible option. Atom $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is in the core because atom $P(\bar{a})$ is not involved in any violation of ICs. If this happen, then rules having $P\_(\bar{a}, \mathbf{f_a})$ in the head have a false body. In this case, $P\_(\bar{a}, \mathbf{f_a})$ is unfounded in $W_{\Pi(D,IC)}$, and as a consequence, it becomes false in $W_{\Pi(D,IC)}$. Given the fact that $P\_(\bar{a}, \mathbf{t}^{\star})$ is in $W^2_{\Pi}(\emptyset)$, and $P\_(\bar{a}, \mathbf{f_a})$ is false, then $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is in $W^3_{\Pi}(\emptyset)$.

   If atom $P(\bar{a})$ is involved in the violation of a IC, then atom $P\_(\bar{a}, \mathbf{t_a})$ has to be in the core, but according to item 2, such atom is never in the core, therefore in this case $P\_(\bar{a}, \mathbf{t}^{\star\star})$ cannot be true in the core.

6. $aux(\bar{a}) \in Core(\Pi(D, IC))$. To prove: $aux(\bar{a}) \in W_{\Pi(D,IC)}$.

   If $aux(\bar{a})$ is true in the core, then a rule of the form $aux(\bar{a}) \leftarrow P\_(\bar{a}, \bar{c}, \mathbf{t}^{\star})$, *not* $P\_(\bar{a}', \bar{c}, \mathbf{f_a})$, $\bar{a} \neq null$, $\bar{c} \neq null$ has a true body in the instantiated program. Atom $P\_(\bar{a}, \bar{c}, \mathbf{t}^{\star})$, is in $W^2_{\Pi}(\emptyset)$ (item 3), since $P\_(\bar{a}', \bar{c}, \mathbf{f_a})$ is false in the core, then there is no rule with a positive body having $P\_(\bar{a}', \bar{c}, \mathbf{f_a})$ in the head. As a consequence, $P\_(\bar{a}', \bar{c}, \mathbf{f_a})$ is declared as false in $W^2_{\Pi}(\emptyset)$, and therefore, $aux(\bar{a})$ is in $W^3_{\Pi}(\emptyset)$.

(b) Since $Core(\Pi(D, IC))$ coincides with set W^+ of the WFI of program $\Pi(D, IC)$, the result follows from Proposition 7.2. □

Theorem 7.1 extends the preliminary results presented in [3], which hold for a set of ICs containing functional dependencies and unary ICs only, to *interaction-free* sets of

ICs, that properly contain those former classes.

Definition 7.7 For a database instance D, a set IC of ICs, and a conjunctive query $\mathcal{Q}$ without projection, a tuple $\bar{t}$ is a *well-founded* answer to $\mathcal{Q}$ if $Ans(\bar{t})$ is in the set W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$. The set of *well-founded* answers to $\mathcal{Q}$ is denoted by $W\!F\!A^+(\mathcal{Q})$. $\Box$

From now on, W^+ is restricted to the atoms annotated with $\mathbf{t}^{\star\star}$ when applied to program $\Pi(D, IC)$, and to the Ans-atoms when applied to program $\Pi(D, IC, \mathcal{Q})$.

Corollary 7.1 For a database instance D, an *interaction-free* set IC of ICs, and queries that are conjunctions of atoms without existential quantifiers:

(a) $Core(\Pi(D, IC))$, and $Core(\Pi(D, IC, \mathcal{Q}))$ can be computed in polynomial time in data complexity (i.e. relative to the size of D).

(b) The sets of core, consistent, and well-founded answers to queries coincide, i.e. $ConsA(\mathcal{Q}) = CoreA(\mathcal{Q}) = W\!F\!A^+(\mathcal{Q})$.

(c) CQA for quantifier free conjunctive queries can be computed in polynomial time.

Proof: (a) The result follows from the fact that the WFI of a disjunctive program can be computed in polynomial time [60]. (b) It follows from Theorem 7.1 and Proposition 7.2. (c) It follows from Theorem 7.1, Proposition 7.2, and the fact that the WFI of a program can be computed in polynomial time [60]. $\Box$

Example 7.6 For $D = \{S(a,b),\ S(a,c),\ S(b,c),\ R(b,c),\ P(a)\ P(b)\}$, and UIC: $\forall xyz(S(x,y),\ S(x,z) \rightarrow y = z)$; and RIC: $\forall x(P(x) \rightarrow \exists y R(x,y))$, the repair program $\Pi(D, IC)$ has four stable models, and $Core(\Pi(D, IC)) = W^+ = \{S\_(b, c, \mathbf{t}^{\star\star}),$ $P\_(b, \mathbf{t}^{\star\star}),\ R\_(b, c, \mathbf{t}^{\star\star})\}$.

Now consider $D = \{S(a,b),\ S(a,c),\ S(b,c),\ R(b,c),\ R(a,b)\}$, and set IC of FDs: $\forall xyz(S(x,y),\ S(x,z) \rightarrow y = z)$; $\forall xyz(R(x,y),\ R(x,z) \rightarrow y = z)$; IND: $\forall xy(S(x,y) \rightarrow R(x,y))$; unary IC: $\forall xy(R(x,y) \rightarrow y = b)$. Program $\Pi(D, IC)$ has one stable model, and $\mathit{Core}(\Pi(D, IC))$= $\{S\_(a,b,\mathbf{t}^{\star\star}),\ R\_(a,b,\mathbf{t}^{\star\star})\}$. However, W^+ does not have any atoms annotated with $\mathbf{t}^{\star\star}$, and $W^+ \subsetneqq \mathit{Core}(\Pi(D, IC))$. This happens because the set of UICs is not *interaction-free*: there is a unary IC on R, and R appears in the consequent of the IND.

For $D = \{S(a,b),\ S(a,c),\ S(b,c),\ R(b,c),\ R(a,b),\ P(a)\ P(b)\}$, and set IC of FDs: $\forall xyz(S(x,y),\ S(x,z) \rightarrow y = z)$; $\forall xyz(R(x,y),\ R(x,z) \rightarrow y = z)$; IND: $\forall xy(S(x,y) \rightarrow R(x,y))$; and RIC: $\forall x(P(x) \rightarrow \exists y R(x,y))$, program $\Pi(D, IC)$ has two stable models. Here, $\mathit{Core}(\Pi(D, IC))$= $\{P\_(a,\mathbf{t}^{\star\star}),\ P\_(b,\mathbf{t}^{\star\star}),\ S\_(b,c,\mathbf{t}^{\star\star}),\ R\_(b,c,\mathbf{t}^{\star\star})\}$, $W^+ = \{P\_(b,\mathbf{t}^{\star\star}),\ S\_(b,c,\mathbf{t}^{\star\star}),\ R\_(b,c,\mathbf{t}^{\star\star})\}$, and $W^+ \subsetneqq \mathit{Core}(\Pi(D, IC))$. This happens because the set of ICs is not *interaction-free*: predicate R is involved both in the RIC and the IND. □

Example 7.7 (example 7.4 cont.) For program $\Pi(D, IC)$, W^+ of the WFI of program $\Pi(D, IC)$ is $\{S\_(b,c,\mathbf{t}^{\star\star})\}$, which coincides with $\mathit{Core}(\Pi(D, IC))$. For query $\mathit{Ans}(x,y) \leftarrow S\_(x,y,\mathbf{t}^{\star\star})$, $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$= $W^+ = \{\mathit{Ans}(b,c)\}$. Hence, the well-founded and the core answer is (b,c), which is also the consistent answer to the query.

For $\mathcal{Q}$: $\mathit{Ans}(y) \leftarrow S\_(x,y,\mathbf{t}^{\star\star})$, $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$= $W^+ = \{\mathit{Ans}(c)\}$, hence the well-founded, the core, and the consistent answer is (c). For $\mathcal{Q}$: $\mathit{Ans} \leftarrow S\_(b,c,\mathbf{t}^{\star\star})$, $S\_(a,b,\mathbf{t}^{\star\star})$, $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$= $W^+ = \emptyset$, therefore the well-founded answer is *no*, as the consistent and core answers to $\mathcal{Q}$.

Nevertheless, for query: $\mathit{Ans}(x) \leftarrow S\_(x,y,\mathbf{t}^{\star\star})$, $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$= $W^+ = \{\mathit{Ans}(b)\}$. Hence, the well-founded and the core answer is (b), but the consistent answers are $(b), (a)$. This happens because the query involves inconsistent tuples that do not fall in a core computation. □

We can see that with the WFI of programs we can compute consistent answers to a restricted set of conjunctive queries with existential quantifiers. Nevertheless, in the following section we show that in presence of FDs (at most one per relation) it is possible to use sets W^+ and W^u of the WFI of programs to retrieve consistent answers to a restricted class of conjunctive queries with existential quantifiers.

It is important to mention that according to the results presented in [27, 39], it is impossible to use a core computation as the WFI of programs to compute consistent answers to all the conjunctive queries with projections. Because, in this case, CQA will be polynomial for this kind of queries.

So far, we have that for *interaction-free* sets of ICs we can compute all the consistent answers to conjunctive queries without projection with the WFI of programs. Moreover, this can be done in polynomial time. As we will show in Example 7.8, for this kind of ICs, we can also use the WFI of programs as a first step to compute consistent answer to ground disjunctive queries. For this kind of queries we cannot ensure that W^+ of $\Pi(D, IC, \mathcal{Q})$ coincides with the core of $\Pi(D, IC, \mathcal{Q})$, but, we can use the WFI as a started point to compute answers. In this manner, the computation of stable models is considered only as a second step, if needed.

Example 7.8 For $D = \{S(a,b),\ R(a,b),\ S(b,c)\}$ and IC: $\forall xy(S(x,y) \rightarrow R(x,y))$, and the disjunctive query: $Ans \leftarrow S\_(a,b,\mathbf{t}^{\star\star}) \lor R\_(b,c,\mathbf{t}^{\star\star})$, which program contains rules: $Ans \leftarrow S\_(a,b,\mathbf{t}^{\star\star})$, and $Ans \leftarrow R\_(b,c,\mathbf{t}^{\star\star})$, W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$ is $\{Ans\}$. Therefore, $\mathit{WFA}^+(\mathcal{Q})$ is *yes*, which coincides with the consistent answer to $\mathcal{Q}$.

For the query: $Ans \leftarrow S\_(b,c,\mathbf{t}^{\star\star}) \lor R\_(b,c,\mathbf{t}^{\star\star})$, which program contains rules: $Ans \leftarrow S\_(b,c,\mathbf{t}^{\star\star})$, and $Ans \leftarrow R\_(b,c,\mathbf{t}^{\star\star})$, W^+ of the WFI of $\Pi(D, IC, \mathcal{Q})$ does not contain any Ans atom. Hence, the well-founded answer is *no*, but the consistent answer is *yes*. This happens because the query involves tuples that are in different database repairs, and they are not captured by a core computation, like the WFI of a program. Therefore, the consistent answer to this query should be obtained by evaluating program $\Pi(D, IC, \mathcal{Q})$ under the stable models semantics. □

Algorithm 7.1 computes consistent answers to ground disjunctive queries in a database instance D, wrt a set IC of *interaction-free* ICs. The input to the algorithm consists of database instance D, the set IC of *interaction-free* ICs, and the query $\mathcal{Q}$. The algorithm first generates program $\Pi(D, IC, \mathcal{Q})$, and it computes the WFI of the program. After that, it tries to answer the query by using W^+ of the WFI of the program $\Pi(D, IC, \mathcal{Q})$. If there are no answers, then answers are computed by evaluating program $\Pi(D, IC, \mathcal{Q})$ under the stable models semantics.

Algorithm 7.1: MIXED COMPUTATION OF CONSISTENT ANSWERS($D, IC, \mathcal{Q}$)

```
Input: database instance $D$, set $IC$ of interaction-free ICs, query $\mathcal{Q}$
Output: consistent answers to $\mathcal{Q}$
$\Pi(D, IC) := GenerateRepairProgram(D, IC)$;
$\Pi(\mathcal{Q}) := GenerateQueryProgram(\mathcal{Q})$;
calculate the WFI of program $\Pi(D, IC, \mathcal{Q})$
if $Ans \in W^+$
  then return ($yes$)
  else { run: $\Pi(D, IC, \mathcal{Q})$ under cautious semantics
         return ($answer$)
```

7.5 Well-Founded Answers with respect to Functional Dependencies

We consider only one functional dependency or key dependency per relation.[7] In this section a FD on a relation P is written with $X \rightarrow Y$, where X and Y are set of attributes of P and $X \cap Y = \emptyset$. The functional dependency on P is satisfied if for two tuples in P that have the same values in attributes in X, they also have the same value for the attributes in Y.

In order to use the WFI of programs for CQA we have to restrict ourselves to the following restricted classes of conjunctive queries.

Definition 7.8 [27] A conjunctive query of the form:

$$Ans(w_1 \ldots w_m) \leftarrow \exists z_1 \ldots z_n (P_1(\bar{x}_1), \ldots P_n(\bar{x}_n)),^{8} \tag{7.1}$$

where $w_1 \ldots w_m, z_1 \ldots z_n$ are all the variables that appear in the atoms of the body of the query, each $\bar{x}_i$ matches the arity of P_i, variables $w_1 \ldots w_m$ are the *free* variables of the query. The query is called *simple* if there are no constants and no repeated symbols in the query (no joins between relations are allowed). □

In particular, we consider conjunctive queries without free variables, i.e. boolean queries of the form:

$$Ans \leftarrow \exists z_1 \ldots z_n (P_1(\bar{x}_1), \ldots P_n(\bar{x}_n)), \tag{7.2}$$

where $z_1 \ldots z_n$ are variables that appear in the atoms of the query, and $\bar{x}_1 \ldots \bar{x}_n$ are variables and/or constants, and no joins between relations are allowed.

[7]A primary key IC can be defined by one or more functional dependencies.

[8]Usually the existential quantifiers are implicit on the query, but in this section, we will write them explicitelly on queries.

In this section we use the concepts of safe database and conflict closure of a database, which were presented in [34] and reviewed in Chapter 5 (cf. Section 5.5). According to Definition 5.7, the safe database S_D is the portion of the database that does not participate in any violation of ICs, and that will never be touched by the repair process. The conflict closure of the database $C_D^\star$ is the set of tuples that violate ICs or are going to be changed to avoid new violations of ICs. We assume that the database domain may contain the null value.

Moreover, we understand satisfaction of ICs as used so far, i.e. a IC is satisfied if any of the relevant attributes has a *null* value, or the IC is satisfied in the traditional way, that is, as first-order satisfaction and with null values treated as any other constant.

We claim that for FDs (at most one per relation), there is a direct relationship between the safe portion of a database instance D, the set W^+ of the WFI of program $\Pi(D, FD)$; and also between the conflict closure of the database and the set W^u of the WFI of the repair program.

Lemma 7.1 For a database instance D, and a set FD of FDs, if an atom of the form $P\_(\bar{c}, \mathbf{t}^{\star\star})$ is in W^+ of the WFI of program $\Pi(D, FD)$, where $\bar{c}$ are constants in $\mathcal{U}$, then $P(\bar{c})$ is in the safe portion of database instance D wrt FD.

Proof: We have to prove that for every atom of the form $P\_(\bar{a}, \mathbf{t}^{\star\star})$ that is in W^+ of the WFI of program $\Pi(D, FD)$, the atom $P(\bar{a})$ is in S_D. This is done in item (a). Also, we need to show that if atom $P(\bar{a})$ is consistent, then $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is in W^+. This is done in item (b).

(a) By contradiction, let us suppose that there exists an atom $P\_(\bar{a}, \mathbf{t}^{\star\star})$ in W^+, but $P(\bar{a})$ is not in S_D.

If atom $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is true in W^+, then according to the interpretation rule $P\_(\bar{a}, \mathbf{t}^{\star\star}) \leftarrow P\_(\bar{a}, \mathbf{t}^{\star}),\ \ not\ P\_(\bar{a}, \mathbf{f_a})$, the atom $P\_(\bar{a}, \mathbf{t}^{\star})$ is true and $P\_(\bar{a}, \mathbf{f_a})$ is false in W^+. If atom $P\_(\bar{a}, \mathbf{t}^{\star})$ is true, then either $P(\bar{a})$ is a database fact, or $P\_(\bar{a}, \mathbf{t_a})$ is true in the program. However, the atom $P\_(\bar{a}, \mathbf{t_a})$ cannot be true in W^+, because we are considering FDs, which are repaired by tuple deletion only. Then, the atom $P\_(\bar{a}, \mathbf{t}^{\star})$ is true because $P(\bar{a})$ is a database fact. If atom $P\_(\bar{a}, \mathbf{f_a})$ is false, then atom $P(\bar{a})$ is not involved in any inconsistency (or is not touched by the repair process). Otherwise, a disjunctive rule in the instantiated program $\Pi(D, FD)$ will have $P\_(\bar{a}, \mathbf{f_a})$ in its head, and $P\_(\bar{a}, \mathbf{f_a})$ will become undetermined. As a consequence, the atom $P(\bar{a})$ is a consistent database fact. But $P(\bar{a})$ is not in S_D. We have reached a contradiction.

(b) By contradiction, let us assume that there exists an atom $P(\bar{a})$ in S_D, but $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is not in W^+.

If $P(\bar{a})$ is in S_D, then $P\_(\bar{a}, \mathbf{t}^{\star})$ is true, and there is no disjunctive rule having $P\_(\bar{a}, \mathbf{f_a})$ in its head, therefore $P\_(\bar{a}, \mathbf{f_a})$ is false. As a consequence, atom $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is a positive consequence of program $\Pi(D, FD)$. We have reached a contradiction. □

Lemma 7.2 For a database instance D, and a set FD of FDs, if an atom of the form $P\_(\bar{c}, \mathbf{t}^{\star\star})$ is in W^u of the WFI of program $\Pi(D, FD)$, where $\bar{c}$ are constants in $\mathcal{U}$, then $P(\bar{c})$ is in the conflict closure of database instance D wrt FD.

Proof: We have to prove that for every atom of the form $P\_(\bar{a}, \mathbf{t}^{\star\star})$ that is in W^u of the WFI of program $\Pi(D, FD)$ there exists an atom $P(\bar{a})$ in $C_D^{\star}$. This in done in item (a). Also, we prove that if $P(\bar{a})$ is in the conflict closure of D, then $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is in W^u. This is done in item (b).

(a) By contradiction, let us assume that there exists an atom $P\_(\bar{a}, \mathbf{t}^{\star\star})$ in W^u, but $P(\bar{a})$ is not in $C^\star_D$.

If atom $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is undetermined then according to rule $P\_(\bar{a}, \mathbf{t}^{\star\star}) \leftarrow P\_(\bar{a}, \mathbf{t}^{\star})$, *not* $P\_(\bar{a}, \mathbf{f_a})$, either atom $P\_(\bar{a}, \mathbf{t}^{\star})$ or atom $P\_(\bar{a}, \mathbf{f_a})$ are undetermined. First, if atom $P\_(\bar{a}, \mathbf{t}^{\star})$ is undetermined, then the atom $P\_(\bar{a}, \mathbf{t_a})$ is undetermined. However, we are considering FDs only, therefore there are no atoms annotated with constant $\mathbf{t_a}$. Therefore, $P\_(\bar{a}, \mathbf{t}^{\star})$ cannot be undetermined.

Hence, the atom $P\_(\bar{a}, \mathbf{f_a})$ is undetermined, which means that atom $P\_(\bar{a}, \mathbf{f_a})$ is in the head of a disjunctive rule whose body is true. Then, according to the WFS no decision can be done about the atoms in the disjunctive head, and as a consequence atoms $P\_(\bar{a}, \mathbf{f_a})$ and $P\_(\bar{a}, \mathbf{t}^{\star\star})$ are undetermined. It is easy to see that $P\_(\bar{a}, \mathbf{f_a})$ is in a disjunctive rule with body true iff there exists a violation of a FD, and $P(\bar{a})$ violates the FD. Then $P(\bar{a})$ is in $C^\star_D$. We have reached a contradiction.

(b) By contradiction, let us assume that there exists an atom $P(\bar{a})$ in $C^\star_D$, but $P\_(\bar{a}, \mathbf{t}^{\star\star})$ is not in W^u.

If atom $P(\bar{a})$ is in $C^\star_D$ then it participates in the violation of a IC or it is affected by the repair process. But, since we are dealing with FDs only, the unique possibility is that atom $P(\bar{a})$ is in D and it violates a FD. Then, there exists a disjunctive rule having atom $P\_(\bar{a}, \mathbf{f_a})$ in its head, and a true body, and therefore according to the WFS the atom $P\_(\bar{a}, \mathbf{f_a})$ is undetermined, and so does atom $P\_(\bar{a}, \mathbf{t}^{\star\star})$. We have reached a contradiction. □

From Lemmas 7.1 and 7.2 we can conclude that for FDs, we can capture the safe and affected database by using W^+ and W^u of the WFI of program $\Pi(D, FD)$, respectively. Therefore, as in [34] we could compute database repairs by computing the repairs for the affected portion of data, and then combining them with the safe portion of data. However, we are not interested in computing repairs, but in retrieving consistent answers to queries. Thus, what we want is to use as much as possible sets W^+ and W^u of the WFI of program $\Pi(D, FD)$, to compute consistent answers to conjunctive queries with projections wrt FDs.

Definition 7.9 For a database instance D, set FD of FDs, and a conjunctive query $\mathcal{Q}$ of the form (7.1), a tuple $\bar{t}$ is a *well-founded answer* to $\mathcal{Q}$ if $Ans(\bar{t})$ is in $W^+ \cup W^u$ of the WFI of program $\Pi(D, FD, \mathcal{Q})$. The set of *well-founded answers* to $\mathcal{Q}$ are denoted by $WFA^{+u}(\mathcal{Q})$. □

As we show in the example below, we cannot use W^+ and W^u of the WFI of programs directly for computing consistent answers to queries, because for some queries we can obtain incorrect answers.

Example 7.9 Consider $\Sigma = \{R(X, Y, Z, W)\}$, FD $X \rightarrow Y$ and $D = \{R(a, b, c, d), R(a, c, c, b), R(b, c, d, e)\}$. For program $\Pi(D, IC)$, W^+ is $\{R\_(b, c, d, e, \mathbf{t}^{\star\star})\}$, and W^u is $\{R\_(a, b, c, d, \mathbf{t}^{\star\star}), R\_(a, c, c, b, \mathbf{t}^{\star\star})\}$. That is, W^+ contains the tuples that are consistent wrt the FD, and W^u the tuples that are not consistent.

For query: $Ans(x) \leftarrow \exists\, yzw\ R(x, y, z, w)$, W^+ of the WFI of program $\Pi(D, FD, \mathcal{Q})$ is $\{Ans(b)\}$, and W^u is $\{Ans(a)\}$, therefore $WFA^{+u}(\mathcal{Q})$ are $(b), (a)$, which coincides with the consistent answers to $\mathcal{Q}$. This works, because the query is retrieving the values for the attribute in the antecedent of the FD, and even though the tuples in W^u are inconsistent, they share the same value for that attribute.

However, for query $Ans(w) \leftarrow \exists\ yzw\ R(x, y, z, w)$, W^+ of the WFI of program $\Pi(D, FD, \mathcal{Q})$ is $\{Ans(e)\}$ and W^u is $\{Ans(d), Ans(b)\}$, therefore $WFA^{+u}(\mathcal{Q})$= $(e), (d), (b)$, but the consistent answer to $\mathcal{Q}$ is (e). This happens because, the query is retrieving the values of an attribute that is not in the antecedent of the FD, and therefore tuples in W^u do not necessarily have the same value for that attribute. □

The previous example suggests that set W^u of the WFI of programs can only be used directly for CQA when the attributes projected in the query are attributes in the antecedent of some FD, because in this case, the inconsistent tuples from W^u will have the same value for that attribute. Therefore, we can use the WFI of programs directly to compute consistent answers to queries of the form (7.1) when each of the free variables on it refer to attributes in the antecedent of a FD on a relation P.

Definition 7.10 Given a relation P with FD $X \rightarrow Y$, the attributes of P are divided into: (a) Antecedent attributes, i.e. the attributes in X. (b) Consequent attributes, i.e. the attributes in Y. (c) Simple attributes, which are attributes neither in X nor Y.

Moreover, for a query $\mathcal{Q}$ of the form (7.1), we say that a free variable w_i with $w_i \in \{w_1 \ldots w_m\}$ refers to an antecedent (respectively, consequent, simple) attribute, if w_i matches a variable in the position of an antecedent (respectively, consequent, simple) attribute on a query predicate P. □

Example 7.10 For the FD: $\mathit{ID} \rightarrow \mathit{Name}$ on relation $R(\mathit{ID}, \mathit{Name}, \mathit{Age})$, the free variable x in query $\mathit{Ans}(x) \leftarrow \exists\ yz\ R(x,y,z)$ refers to the attribute ID, which is in the antecedent of the FD. For query $\mathit{Ans}(y) \leftarrow \exists\ xz\ R(x,y,z)$, the free variable y refers to an attribute in the consequent of the FD. For query $\mathit{Ans}(z) \leftarrow \exists\, xy\ R(x,y,z)$, the free variable z refers to a simple attribute. □

Proposition 7.4 For a database instance D, set FD of FDs (at most one functional or key dependency per relation), if the free variables in query $\mathcal{Q}$ of the form (7.1) refer to antecedent attributes, then the consistent answers to $\mathcal{Q}$ wrt FD coincide with the well-founded answers to $\mathcal{Q}$.

Proof: We need to prove that if a tuple $\bar{t}$ is a consistent answers to $\mathcal{Q}$, then $Ans(\bar{a})$ is either in W^+ or in W^u of $\Pi(D, FD, \mathcal{Q})$, and viceversa.

There are three cases to evaluate. First, we need to show that for every tuple $\bar{a}$ that is a consistent answer to $\mathcal{Q}$, there exists an atom $Ans(\bar{a})$ in W^+ or in W^u. Second, if there exists an atom $Ans(\bar{a})$ in W^+, then $\bar{a}$ is a consistent answer to $\mathcal{Q}$. Finally, we need to show that if there exists an atom $Ans(\bar{a})$ in W^u, then $\bar{a}$ is a consistent answer to $\mathcal{Q}$.

(a) If tuple $\bar{a}$ is a consistent answer to $\mathcal{Q}$, then $Ans(\bar{a})$ is in W^+ or W^u.

There are two cases to analyze. First, the instantiated atoms of the form $P\_(\bar{c}, \mathbf{t}^{\star\star})$ on the query rule are not involved in any inconsistency wrt FD. In this case, the atoms $P\_(\bar{c}, \mathbf{t}^{\star\star})$ and $Ans(\bar{a})$ are trivially in W^+ of $\Pi(D, FD, \mathcal{Q})$. Second, some of the atoms $P\_(\bar{c}, \mathbf{t}^{\star\star})$ (could be all of them) participate in the violation of FDs in FD. If atom $P(\bar{c})$ is inconsistent wrt FD, then there exists another atom $P(\bar{b})$ such that $P(\bar{c})$ and $P(\bar{b})$ together violate FD. Then, atoms of the form $P\_(\bar{b}, \mathbf{t}^{\star\star})$ and $P\_(\bar{c}, \mathbf{t}^{\star\star})$ are in W^u of $\Pi(D, FD, \mathcal{Q})$, and therefore $Ans(\bar{a})$ is in W^u.

(b) If atom $Ans(\bar{a})$ is in W^+ of $\Pi(D, FD, \mathcal{Q})$, then $\bar{a}$ is a consistent answer to $\mathcal{Q}$.

Since $Ans(\bar{a})$ is in W^+, the instantiated query rule is satisfied in the program, and every atom in the body of the rule is consistent wrt FD. Otherwise, some atom $P\_(\bar{b}, \mathbf{t}^{\star\star})$ in the query will be in the head of a disjunctive rule, and in this case $P\_(\bar{b}, \mathbf{t}^{\star\star})$ would be undetermined and also atom $Ans(a)$. Therefore, tuple $\bar{a}$ is a consistent answer to $\mathcal{Q}$.

(c) If atom $Ans(\bar{a})$ is in W^u of $\Pi(D, FD, \mathcal{Q})$, then $\bar{a}$ is a consistent answer to $\mathcal{Q}$.

If $Ans(\bar{a})$ is in W^u, then some of the atoms in the instantiated query rule is inconsistent wrt FD. Therefore, there exists at least two atoms of the form

$P(\bar{b})$, $P(\bar{c})$ with $\bar{a} \cap (\bar{b} \cup \bar{c}) \neq \emptyset$ that together violate FD, and as a consequence atoms $P\_(\bar{b}, \mathbf{t}^{\star\star})$ and $P\_(\bar{c}, \mathbf{t}^{\star\star})$ are in W^u. Since we are considering only one FD per relation (or a key dependency), a database repair will have either $P(\bar{b})$ or $P(\bar{c})$. Hence, since the free variables in the query refer to antecedent attributes only, $Ans(\bar{a})$ will be true in every repair, and hence $\bar{a}$ is a consistent answer. □

As shown in Example 7.9, when the free variables in a query of the form (7.1) do not refer to attributes in the antecedent of FDs, we cannot use set W^u of $\Pi(D, FD, \mathcal{Q})$ directly to compute consistent answers to queries with projections. However, we can rewrite queries in such a way, that when they are evaluated on W^u of $\Pi(D, FD)$, i.e. the repair program alone, they only retrieve consistent answers. We illustrate this in the example below.

Example 7.11 For $\Sigma = \{R(X, Y, Z, W)\}$, FD: $Z \rightarrow W$, and $D = \{R(a, b, c, d)$, $R(a, b, c, e)$, $R(b, c, d, e)\}$. For the repair program $\Pi(D, FD)$, $W^+ = \{R\_(b, c, d, e, \mathbf{t}^{\star\star})\}$, and $W^u = \{R\_(a, b, c, d, \mathbf{t}^{\star\star})$, $R\_(a, b, c, e, \mathbf{t}^{\star\star})\}$. There are two repairs: $\{R(b, c, d, e)$, $R(a, b, c, d)\}$ and $\{R(b, c, d, e)$, $R(a, b, c, e)\}$.

For $\mathcal{Q}$: $Ans(z, w) \leftarrow \exists xy R(x, y, z, w)$, the free variable z refers to the attribute in the antecedent of the FD, and variable w refers to its consequent. For $\Pi(D, FD, \mathcal{Q})$ it holds $W^+ = \{Ans(d, e)\}$, and $W^u = \{Ans(c, d), Ans(c, e)\}$. Therefore, $WFA^{+u}(\mathcal{Q})$ are (d, e), (c, d), (c, e), but the consistent answer to $\mathcal{Q}$ is (d, e). Thus, in this case, W^u of $\Pi(D, FD, \mathcal{Q})$ does not provide consistent answer to the query. However, we can filter the inconsistent tuples from W^u of $\Pi(D, FD)$ by evaluating the following query on it:

$$\mathcal{Q}' : Ans(z, w) \leftarrow \exists xy R\_(x, y, z, w, \mathbf{t}^{\star\star}) \land \forall x'y'w'(R\_(x', y', z, w', \mathbf{t}^{\star\star}) \rightarrow w' = w).$$

When $\mathcal{Q}'$ is evaluated on W^u of $\Pi(D, FD)$, the answer is empty. Therefore, the final answer to $\mathcal{Q}$ is (d, e), which coincides with the consistent answer to $\mathcal{Q}$.

For query $\mathcal{Q}$: $Ans(z, x) \leftarrow \exists yw R(x, y, z, w)$, where the free variable z refers to the attribute in the antecedent of the FD, and variable x refers to a simple attribute, W^+ of $\Pi(D, FD, \mathcal{Q})$ is $\{Ans(d, b)\}$. Since z refers to an antecedent attribute, we know that tuples in W^u that are inconsistent wrt FD will share the value for that attribute. Thus, we just need to ensure that tuples from W^u have the same value for the attribute referenced by variable x. Therefore, $\mathcal{Q}'$ is:

$$\mathcal{Q}' : Ans(z, x) \leftarrow \exists yw R\_(x, y, z, w, \mathbf{t}^{\star\star}) \wedge \forall x'y'w'(R\_(x', y', z, w', \mathbf{t}^{\star\star}) \rightarrow x' = x).$$

The answer to $\mathcal{Q}'$ evaluated on W^u of $\Pi(D, FD)$ is (c, a). Therefore, the final well-founded answers to $\mathcal{Q}$ are $(d, b), (c, a)$, which coincide with the consistent answers to $\mathcal{Q}$.

Moreover, for $\mathcal{Q}$: $Ans(y) \leftarrow \exists xzw\ R\_(x, y, z, w)$, where variable y refers to a simple attribute, W^+ of $\Pi(D, FD, \mathcal{Q})$ is $\{Ans(c)\}$. Here we just need to ensure that the value for the attribute referenced by variable y is the same in every inconsistent tuple in W^u. Hence, $\mathcal{Q}'$ is:

$$\mathcal{Q}' : Ans(y) \leftarrow \exists xzw R\_(x, y, z, w, \mathbf{t}^{\star\star}) \wedge \forall x'y'z'w'(R\_(x', y', z', w', \mathbf{t}^{\star\star}) \rightarrow \bar{y}' = y).$$

The answer to $\mathcal{Q}'$ evaluated on W^u of $\Pi(D, FD)$ is (b). Therefore, the well-founded answers to $\mathcal{Q}$ are $(c), (b)$, which coincide with the consistent answers to $\mathcal{Q}$. □

The situation is not different for boolean conjunctive queries of the form (7.2). For this kind of queries we also need a new, rewritten query in order to retrieve consistent answers from the set of undetermined atoms of the WFI of a repair program. However, the new query is only needed when it is not possible to compute an answer from W^+ of $\Pi(D, FD, \mathcal{Q})$. We illustrate this in the example below.

Example 7.12 (example 7.11 cont.) For $\mathcal{Q}$: $Ans \leftarrow \exists xzw\ R(x, c, z, w)$, W^+ of $\Pi(D, FD, \mathcal{Q})$ is $\{Ans\}$, therefore the well-founded answer is *yes*, which coincides with consistent answer. For $\mathcal{Q}$: $Ans \leftarrow \exists yzw\ R(a, y, z, w)$, W^+ of $\Pi(D, FD, \mathcal{Q})$ does not have an Ans-atom, then we need to check if we can obtain an answer from W^u of the repair program. The rewritten query is:

$$\mathcal{Q}' : Ans \leftarrow \exists yzw R\_(a, y, z, w, \mathbf{t}^{\star\star}) \wedge \forall x'y'z'w'(R\_(x', y', z', w', \mathbf{t}^{\star\star}) \rightarrow x' = a).$$

When query $\mathcal{Q}'$ is evaluated on $W^u = \{R\_(a, b, c, d, \mathbf{t}^{\star\star}),\ R\_(a, b, c, e, \mathbf{t}^{\star\star})\}$, the answer is *yes*, since every inconsistent tuple wrt the FD has the same value for the first attribute. Therefore, the well-founded answer is *yes*, and coincides with the consistent answer to $\mathcal{Q}$.

Moreover, for $\mathcal{Q}$: $Ans \leftarrow \exists xyz\ R(x, y, z, f)$, there is no Ans-atom in W^+ of $\Pi(D, FD, \mathcal{Q})$, then the following rewritten query is generated:

$$\mathcal{Q}' : Ans \leftarrow \exists xyz R\_(x, y, z, f, \mathbf{t}^{\star\star}) \wedge \forall x'y'z'w'(R\_(x', y', z', w', \mathbf{t}^{\star\star}) \rightarrow w' = f.)$$

However, the answer to $\mathcal{Q}'$ is also negative in W^u, hence the well-founded answer to $\mathcal{Q}$ is *no*, which coincides with the consistent answer to $\mathcal{Q}$. □

The rewriting of queries we introduced in Examples 7.11 and 7.12 corresponds to the rewriting method presented in [39], where it is used to compute consistent answers wrt primary key ICs, to conjunctive queries with existential quantifiers. This method works for a more general case of conjunctive queries, the *C-Tree* queries, which allow joins between different database relations.

In [39] the rewritten query is evaluated directly on the inconsistent database instance. In this manner, its answers correspond to the consistent answers to the original query. Here, the rewritten query filters inconsistent answers from set W^u of the WFI of a repair program. Thus, the rewritten query is evaluated on a subset of the database, the portion of data that falls in W^u. Therefore, we compute rewritten queries on small portions of the database, instead of processing them on the original database.

Algorithm 7.2: REWRITTEN QUERY GENERATION($FD, \mathcal{Q}$)

```
Input: query $\mathcal{Q}$, and set $FD$ of FDs
Output: rewritten query $\mathcal{Q}_{rew}$
Variables: $FANS$, $AV$, $CV$, $AVP$, $CVP$: set of variables;
  $\Pi(\mathcal{Q}) := GenerateQueryProgram(\mathcal{Q})$;
  $\mathcal{Q}_{rew} := \Pi(\mathcal{Q})$;
  $FANS := IdentifyFreeVariables(\mathcal{Q})$;
  $AV := IdentifyVariablesInAntecedentsFDs(\mathcal{Q}, FD)$;
  $CV := IdentifyVariablesInConsequentsFDs(\mathcal{Q}, FD)$;
  for each $P(\bar{x}) \in \mathcal{Q}$
    do
      $AVP := \bar{x} \cap AV$;   $CVP := \bar{x} \cap CV$;   $FVP := \bar{x} \cap FANS$;
      if $(\{AVP \cup CVP \cup FVP\} \neq \emptyset)$
        then
          $\bar{y} := \bar{x} \smallsetminus AVP$;
          $\bar{y} := GenerateFreshVariables(\bar{y})$;
          $\mathcal{Q}_{rew} := \mathcal{Q}_{rew} \wedge \forall\bar{y}\, GenerateAtom(P(\bar{x}), \bar{y}, AVP, CVP, FVP)$;
return $(\mathcal{Q}_{rew})$
```

Algorithm 7.2 generates a rewritten query $\mathcal{Q}_{rew}$ for a given query $\mathcal{Q}$ of the form (7.1). The input to the algorithm consists of the set FD of FDs, and the query $\mathcal{Q}$. The algorithm first identifies the free variables in $\mathcal{Q}$. After that, it determines which variables refer to attributes in FDs. Then, for each atom in the query that share variables with the *Ans* predicate, an atom in the rewritten query is generated. A tuple $\bar{t}$ is an answer to the rewritten query if tuples from set W^u of $\Pi(D, FD)$ have the same values for the attributes referenced by the free variables in the query.

For boolean conjunctive queries of the form (7.2) the rewritten query is only generated when the original query cannot be answered with set W^+ of $\Pi(D, FD, \mathcal{Q})$. Algorithm 7.3 generates a rewritten query $\mathcal{Q}_{rew}$ for a query $\mathcal{Q}$ of the form (7.2). The algorithm first identifies the query predicates that have instantiated variables. Then for each of those query atoms, the algorithm generates a corresponding atom in the rewritten query. The answer to $\mathcal{Q}_{rew}$ will be *yes* if tuples from W^u of $\Pi(D, FD)$ have the same values for the instantiated variables in the original query.

Algorithm 7.3: Rewritten Query Generation($\mathcal{Q}$)

```
Input: query $\mathcal{Q}$
Output: rewritten query $\mathcal{Q}_{rew}$
Variables: $CONS$ : set of constants;
  $\Pi(\mathcal{Q}) := GenerateQueryProgram(\mathcal{Q})$;
  $\mathcal{Q}_{rew} := \Pi(\mathcal{Q})$;
  $CONS := IdentifyConstants(\mathcal{Q})$;
  for each $P(\bar{x}) \in \mathcal{Q}$
    do
      if $(\{CONS \cap \bar{x}\} \neq \emptyset)$
        then
          $\bar{y} := \bar{x} \smallsetminus \{CONS \cap \bar{x}\}$;
          $\bar{y} := GenerateFreshVariables(\bar{y})$;
          $\mathcal{Q}_{rew} := \mathcal{Q}_{rew} \wedge \forall \bar{y} GenerateAtom(P(\bar{x}), \bar{y}, CONS)$;
return $(\mathcal{Q}_{rew})$
```

It is important to notice that when the ICs are FDs only (at most one functional or key dependency per relation), we can also use W^u of $\Pi(D, FD, \mathcal{Q})$ to obtain consistent answers to ground disjunctive queries. This is illustrated in the example below.

Example 7.13 (example 7.9 cont.) For query: $Ans \leftarrow R\_(a, b, c, d, \mathbf{t}^{\star\star}) \vee R\_(b, c, d, e, \mathbf{t}^{\star\star})$, which program contains rules: $Ans \leftarrow R\_(a, b, c, d, \mathbf{t}^{\star\star})$, and $Ans \leftarrow R\_(b, c, d, e, \mathbf{t}^{\star\star})$, atom Ans is in W^+ of $\Pi(D, FD, \mathcal{Q})$. Therefore, the well-founded answer to $\mathcal{Q}$ is *yes*, which coincides with the consistent answer to $\mathcal{Q}$.

For $\mathcal{Q}$: $Ans \leftarrow R\_(a, b, c, d, \mathbf{t}^{\star\star}) \vee R\_(a, b, c, e, \mathbf{t}^{\star\star})$, which program contains rules: $Ans \leftarrow R\_(a, b, c, d, \mathbf{t}^{\star\star})$, and $Ans \leftarrow R\_(a, b, c, e, \mathbf{t}^{\star\star})$, atom Ans is not in W^+, but it is in W^u of $\Pi(D, FD, \mathcal{Q})$. Therefore, the well-founded answer is *yes*, and also coincides with the consistent answer to $\mathcal{Q}$. □

Definition 7.11 For a database instance D, and set FD of FDs (at most one functional or key dependency per relation), the consistent answer to a ground disjunctive query $\mathcal{Q}$ wrt FD is *yes* if Ans is in $W^+ \cup W^u$ of $\Pi(D, FD, \mathcal{Q})$. Otherwise, the consistent answer to $\mathcal{Q}$ is *no*. □

The use of the WFI of programs to compute consistent answers wrt FDs (at most one per relation) is relevant. This is because, for conjunctive queries of forms (7.1) and (7.2) CQA can be computed in polynomial time.

Proposition 7.5 For a database instance D, and a set FD of FDs (at most one functional or key dependency per relation), the consistent answers to a query that is either a possibly existentially quantified conjunctive query of the forms (7.1), (7.2), or a ground disjunctive queries can be computed in polynomial time.

Proof: It follows from the fact that the WFI of a program can be computed in polynomial time [60]. □

In [27], a polynomial time rewriting method retrieves consistent answers, regarding FDs (at most one per relation), for the class of closed simple conjunctive queries, which are queries of the form (7.2). The authors use graph representations for database repairs. The first-order query rewriting method presented in [39] also computes, in polynomial time, consistent answers wrt primary key constraints for conjunctive queries with existential quantifiers. This method works for a more general case of conjunctive queries, the *C-Tree* queries, which allow joins between different database relations.

We adopt the rewritten method presented in [39] for computing consistent answers to the restricted set of conjunctive queries of the forms (7.1), and (7.2). However, we compute the rewritten query on a subset of the database, the portion of data that falls in W^u of program $\Pi(D, FD)$. Therefore, we compute rewritten queries on small portions of the database, instead of processing them on the original database.

7.6 Well-Founded Semantics as an Approximation to CQA

For RIC-acyclic sets of ICs, W^+ of $\Pi(D, IC)$ is a subset of the intersection of their stable models. Therefore, the WFI of programs will not retrieve all the consistent answers to positive Datalog queries, i.e. conjunctive or disjunctive queries with projection but without negation. However, for Positive Datalog queries, we can ensure that the well-founded answers, those obtained from W^+ of $\Pi(D, IC, \mathcal{Q})$, are a subset of the consistent answers to queries. We call to these answers *approximate consistent answers.*

To obtain approximate answers may be important if, for example, we are not looking for all the consistent answers, but we want to now if the queries have some consistent answer. The approximate consistent answers can be computed in polynomial time, and as we show in Section 7.7 we can obtain answers from W^+ efficiently in the XSB system [25].

Definition 7.12 For a given database instance D, a RIC-acyclic set IC of ICs, and a positive Datalog query $\mathcal{Q}$, a tuple $\bar{t}$ is an *approximate consistent answer* to $\mathcal{Q}$ if $Ans(\bar{t})$ is in W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$.[9] $\square$

Thus, the set of *approximate consistent answers* to a query $\mathcal{Q}$ is a subset of the set of consistent answers to $\mathcal{Q}$. We illustrate this in the example below.

Example 7.14 (example 7.6 cont.) For $D = \{S(a,b), S(a,c), S(b,c), R(b,c), R(a,b), P(a)\ P(b)\}$, and set IC of FDs: $\forall xyz(S(x,y), S(x,z) \rightarrow y = z)$; $\forall xyz(R(x,y), R(x,z) \rightarrow y = z)$; IND: $\forall xy(S(x,y) \rightarrow R(x,y))$; and RIC: $\forall x(P(x) \rightarrow \exists y R(x,y))$. Program $\Pi(D, IC)$ has two stable models:

$$\begin{aligned}
\mathcal{M}_1 \;=\; & \{S\_(a,b,\mathbf{t}^\star), S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), P\_(a,\mathbf{t}^\star), P\_(b,\mathbf{t}^\star),\\
& aux(b), aux(a), \underline{P\_(a,\mathbf{t}^{\star\star})}, \underline{P\_(b,\mathbf{t}^{\star\star})}, R\_(a,c,\mathbf{t_a}), S\_(a,b,\mathbf{f_a}), \underline{S\_(a,c,\mathbf{t}^{\star\star})},\\
& \underline{S\_(b,c,\mathbf{t}^{\star\star})}, R\_(a,b,\mathbf{f_a}), \underline{R\_(b,c,\mathbf{t}^{\star\star})}, R\_(a,c,\mathbf{t}^\star), \underline{R\_(a,c,\mathbf{t}^{\star\star})}\},\\
\mathcal{M}_2 \;=\; & \{S\_(a,b,\mathbf{t}^\star), S\_(a,c,\mathbf{t}^\star), S\_(b,c,\mathbf{t}^\star), R\_(a,b,\mathbf{t}^\star), R\_(b,c,\mathbf{t}^\star), P\_(a,\mathbf{t}^\star), P\_(b,\mathbf{t}^\star),\\
& aux(b), aux(a), \underline{P\_(a,\mathbf{t}^{\star\star})}, \underline{P\_(b,\mathbf{t}^{\star\star})}, S\_(a,c,\mathbf{f_a}), \underline{S\_(a,b,\mathbf{t}^{\star\star})}, \underline{S\_(b,c,\mathbf{t}^{\star\star})},\\
& \underline{R\_(a,b,\mathbf{t}^{\star\star})}, \underline{R\_(b,c,\mathbf{t}^{\star\star})}\}.
\end{aligned}$$

Here, $Core(\Pi(D, IC))$= $\{P\_(a,\mathbf{t}^{\star\star}),\ P\_(b,\mathbf{t}^{\star\star}),\ S\_(b,c,\mathbf{t}^{\star\star}),\ R\_(b,c,\mathbf{t}^{\star\star})\}$, W^+ of $\Pi(D, IC)$ is $\{P\_(b,\mathbf{t}^{\star\star}), S\_(b,c,\mathbf{t}^{\star\star}), R\_(b,c,\mathbf{t}^{\star\star})\}$, and $W^+ \subsetneq Core(\Pi(D, IC))$. This happens because the set of ICs is not *interaction-free*: predicate R is involved in the RIC and in the IND.

[9]Here set W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$ is a subset of $Core(\Pi(D, IC, \mathcal{Q}))$.

For query $Ans(x) \leftarrow P(x)$, $Core(\Pi(D, IC, \mathcal{Q}))= \{Ans(a), Ans(b)\}$, but W^+ of $\Pi(D, IC, \mathcal{Q})$ is $\{Ans(b)\}$. Therefore, the well-founded answer is (b), but the consistent answers are $(a), (b)$. Thus, with set W^+ we obtain an approximate answer set to the query.

For the disjunctive query $Ans(x) \leftarrow P(x) \vee R(x, y)$, that as a program becomes: $Ans(x) \leftarrow P\_(x, \mathbf{t}^{\star\star})$, and $Ans(x) \leftarrow R\_(x, y, \mathbf{t}^{\star\star})$. Again, $Core(\Pi(D, IC, \mathcal{Q}))= \{Ans(a), Ans(b)\}$, but W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$ is $\{Ans(b)\}$. Thus, the well-founded answer (b) is an approximate answer to $\mathcal{Q}$. □

In this manner, consistent answers to queries are computed by focalizing on W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$, which contains a subset of the *Ans*-atoms that belong to every stable model of $\Pi(D, IC, \mathcal{Q})$. Therefore, for positive Datalog queries we obtain a subset of the consistent answers to queries. Nevertheless, we have a polynomial time approximation algorithm for CQA.

Corollary 7.2 For a database instance D, a RIC-acyclic set IC of ICs, and a positive Datalog query $\mathcal{Q}$, i.e. a conjunctive or disjunctive query with projection but without negation, the well-founded answers to $\mathcal{Q}$, obtained from W^+ of $\Pi(D, IC, \mathcal{Q})$, are a subset of the consistent answers to $\mathcal{Q}$. In symbols, $WFA^+(\mathcal{Q}) \subseteq ConsA(\mathcal{Q})$.[10]

Proof: It follows from the fact that for a disjunctive program Π, $W_{(\Pi)} \subseteqq Core(\Pi)$ [60]. □

It is important to remark that we cannot use W^+ of $\Pi(D, IC, \mathcal{Q})$ alone for computing consistent answers to Datalog queries with negation. We illustrate this in the example below.

[10] As defined in Definition 7.7, $WFA^+(\mathcal{Q})$ is the set of *well-founded* answers to $\mathcal{Q}$ obtained from W^+ of $\Pi(D, IC, \mathcal{Q})$.

Example 7.15 (example 7.14 cont.) For query $Ans \leftarrow$ *not* $P(a)$, W^+ of $\Pi(D, IC, \mathcal{Q})$ is $\emptyset$, but we cannot ensure that the consistent answer is *no*. Actually, atom Ans is in W^u of $\Pi(D, IC, \mathcal{Q})$, therefore the answer to this query will be *undefined* under the well-founded semantics.[11] $\square$

Notice that, since for *interaction-free* sets of RIC-acyclic ICs, and conjunctive queries without projection, $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$ coincides with W^+ of $\Pi(D, IC, \mathcal{Q})$ (cf. Theorem 7.1), the set of *approximate consistent answers* to this kind of queries coincides with the set of consistent answers.

7.7 Computing Well-Founded Answers to Queries

The XSB system is a logic programming and deductive database system, which uses SLG resolution [25] to compute queries to normal programs (programs without disjunction) under the WFS. In [9] it was shown that a *head-cycle free* disjunctive program can be translated into an equivalent normal program.

Definition 7.13 For a given disjunctive program Π, a dependency graph is constructed as follows: each literal in $\mathit{ground}(\Pi)$ is a node in the graph, and there is an edge from L to L' iff there is a rule r in which L appears positive in the body, and L' appears in the head of r. Π is *head-cycle free* (HCF) if the graph does not contain directed cycles that go through two literals that belong to the head of the same rule. $\square$

A HCF disjunctive program Π can be transformed into a non-disjunctive program $sh(\Pi)$ with the same stable models [9], that is obtained by replacing every disjunctive

[11]In this section, we are analyzing the computation of well-founded answers from W^+ of the WFI of program $\Pi(D, IC, \mathcal{Q})$, only.

rule on Π of the form

$$\bigvee_{i=1}^{n} p_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^{m} q_j(\bar{y}_j),$$

by the n rules:

$$p_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^{m} q_j(\bar{y}_j) \wedge \bigwedge_{k \neq i} \; not \; p_k(\bar{x}_k),$$

where $i = 1, \ldots, n$.

HCF programs have better computational complexity than general disjunctive programs. In fact, the computational complexity of skeptical (or cautious) query evaluation for HCF programs is co-NP-complete, but for general disjunctive programs is Π_2^P-complete [30].

In [7] it was proven that if the ICs are only of the form $\bigwedge_{i=1}^{n} P_i \; (\bar{x}_i) \rightarrow \varphi$, where $P_i(\bar{x}_i)$ is an atom and φ is a formula containing built-ins, the corresponding repair program is always HCF.[12] Notice that this is also valid for NNCs. Therefore, for HCF programs, the well-founded answers to queries can be computed directly in the XSB system.

XSB evaluates queries based in the WFI of the program, but when a program is evaluated in the XSB system, it does not show explicitly the whole sets W^+, W^u or W^- of the WFI of the program. For instance, for a ground conjunctive queries XSB returns either: *yes, no* or *undefined* as an answer; for conjunctive queries with free variables, XSB computes one by one the *true* and *undefined* answers. The consistent answers would be the answers that are defined as *true.* Hence, the complete set of answers can be obtained by doing backtracking.[13]

Example 7.16 (example 7.1 cont.) For the database instance $D = \{S(a, b),\, S(a, c),\, S(b, c)\}$ and IC: $\forall xyz(S(x, y) \wedge S(x, z) \rightarrow y = z)$, the repair program $\Pi(D, IC)$ is

[12]That includes important classes of UICs such as, key constraints, and functional dependencies.

[13]In XSB, backtracking can be executed by typing any key on the keyboard different from the "Enter" key.

HCF, and therefore it can be translated into the following non-disjunctive program:

$S(a,b).\quad S(a,c).\quad S(b,c).$

$S\_(x,y,\mathbf{f_a}) \leftarrow S\_(x,y,\mathbf{t}^\star), S\_(x,z,\mathbf{t}^\star), y \neq z, x \neq null, y \neq null, z \neq null,\ not\ S\_(x,z,\mathbf{f_a}).$

$S\_(x,z,\mathbf{f_a}) \leftarrow S\_(x,y,\mathbf{t}^\star), S\_(x,z,\mathbf{t}^\star), y \neq z, x \neq null, y \neq null, z \neq null,\ not\ S\_(x,y,\mathbf{f_a}).$

$S\_(x,y,\mathbf{t}^\star) \leftarrow S(x,y).$

$S\_(x,y,\mathbf{t}^\star) \leftarrow S\_(x,y,\mathbf{t_a}).$

$S\_(x,y,\mathbf{t}^{\star\star}) \leftarrow S\_(x,y,\mathbf{t}^\star),\ not\ S\_(x,y,\mathbf{f_a}).$

For $\mathcal{Q}$: $Ans(x,y) \leftarrow S\_(x,y,\mathbf{t}^{\star\star})$, XSB returns $x=b,\ y=c$, as true answer to the XSB query obtained from program $\Pi(D, IC, \mathcal{Q})$.[14] This well-founded answer coincides with the consistent answer to $\mathcal{Q}$.

For $\mathcal{Q}$: $Ans(x) \leftarrow S\_(x,y,\mathbf{t}^{\star\star})$, XSB returns $x = b$ as true answer, but the consistent answers are $(b), (a)$. This happens because the query involves tuples $\{S(a,b), S(a,c)\}$ which belong to different repairs, and therefore they are not captured by the WFI of program $\Pi(D, IC, \mathcal{Q})$.

Moreover, for the ground disjunctive query: $Ans \leftarrow S\_(a,b,\mathbf{t}^{\star\star}) \vee S\_(a,c,\mathbf{t}^{\star\star})$, which is expressed by the two non-disjunctive rules: $Ans \leftarrow S\_(a,b,\mathbf{t}^{\star\star})$, and $Ans \leftarrow S\_(a,c,\mathbf{t}^{\star\star})$, XSB returns *undefined* as an answer, but the consistent answer is *yes*. Again, this happens because the tuples involved in the query are in different repairs, and as a consequence, they cannot be fetched in W^+ of the program.

However, for the ground disjunctive query $Ans \leftarrow S\_(b,c,\mathbf{t}^{\star\star}) \vee S\_(a,c,\mathbf{t}^{\star\star})$, which is expressed by: $Ans \leftarrow S\_(b,c,\mathbf{t}^{\star\star})$ and $Ans \leftarrow S\_(a,c,\mathbf{t}^{\star\star})$, XSB returns *yes* as an answer, which coincides with the consistent answer to $\mathcal{Q}$. $\square$

In addition, well-founded answers to queries can also be computed by using the *deterministic set* of a (disjunctive) logic program, which is efficiently computed in the DLV

[14] In the XSB system queries are entered as rules without head. For instance, in order to obtain the answers to query $Ans(x,y) \leftarrow S\_(x,y,\mathbf{t}^{\star\star})$, we have to add the query $: -Ans(x,y)$. The answers are obtained by doing backtracking.

system [61].[15] This set corresponds to the atoms that can be deterministically inferred from a program given a certain interpretation. This set, $\mathit{det}(\Pi)$, is contained in every stable model of the program, and can be computed in polynomial time [19]. In fact, for a program Π, it holds that $W^+ \subseteq \mathit{det}(\Pi) \subseteq \bigcap\{S \mid S \text{ is a stable model of program } \Pi\}$, where W^+ is the set of positive atoms in the WFI of program Π [19].

In particular, from Theorem 7.1 we can conclude that for a database instance D, an *interaction-free* set IC of ICs, and query $\mathcal{Q}$ that is a conjunctive query without existential quantifiers, W^+ of $\Pi(D, IC, \mathcal{Q})$, $\mathit{det}(\Pi(D, IC, \mathcal{Q}))$ and $\mathit{Core}(\Pi(D, IC, \mathcal{Q}))$ coincide, where $\mathit{det}(\Pi(D, IC, \mathcal{Q}))$ is the deterministic set of program $\Pi(D, IC, \mathcal{Q})$ restricted to the Ans-atoms. Therefore, the deterministic set becomes an important tool to be used in CQA for non-existentially quantified conjunctive queries.

Example 7.17 (example 7.16 cont.) For query: $\mathit{Ans}(x, y) \leftarrow S\_(x, y, \mathbf{t}^{\star\star})$, the deterministic set of program $\Pi(D, IC, \mathcal{Q})$ is $\{\mathit{Ans}(b, c)\}$, therefore the answer is (b, c), which coincides with the consistent, and the well-founded answers to $\mathcal{Q}$. □

7.8 Summary

In this chapter we proved that for *interaction-free* sets of ICs, the core of a repair program $\Pi(D, IC)$, and the core of program $\Pi(D, IC, \mathcal{Q})$ for queries that are conjunction of atoms without existential quantifiers coincide with the true atoms in the WFI of the respective programs. This is relevant because, since the WFI is computed in polynomial time, core computations for CQA become also polynomial.

In addition, we showed that in the presence of FDs (at most one FD or key dependency per relation) it is possible to use the set of undetermined atoms of the WFI of programs to CQA. In fact, by using both sets W^+ and W^u, we can compute

[15]The deterministic set of a program can be captured in the DLV system by running the program with option **-det**.

all the consistent answers for a restricted set of conjunctive queries with existential quantifiers.

The method presented in [27], where database repairs are specified by graph representations, also retrieves consistent answers regarding FDs (at most one per relation) for the class of closed simple conjunctive queries of the form (7.2). Moreover, the first-order query rewriting method in [39] also computes, in polynomial time, consistent answers wrt primary key constraints for a more general set of conjunctive queries with existential quantifiers. Query rewriting in [39] coincides with the rewriting presented here for the restricted set of conjunctive queries of the forms (7.1) and (7.2).

It is relevant to mention that the set of true atoms of the WFI of a program can be used as a first step to compute consistent answers to ground disjunctive queries wrt *interaction-free* sets of ICs. In this manner, the computation of stable models is left as a second option, if needed. Moreover, we showed that if the WFI of a program is used as a unique way to compute consistent answers, for positive Datalog queries, we may retrieve only a subset of the consistent answers. Nevertheless, we have a polynomial time algorithm for CQA. This is relevant given the high data complexity of CQA (in general CQA is Π^2_p-complete in data complexity [17]).

We can compute well-founded answers using the XSB system, for the restricted class of HCF programs. The XSB system does not show the sets W^+, W^u or W^- of the WFI of the program. But, it evaluates queries based in the WFI of the program. For instance, for conjunctive queries with free variables, XSB computes one by one the *true* and *undefined* answers. The *true* answers are those that fall in W^+ of the WFI of the program, and the *undefined* answers are those in W^u.

Moreover, when the set of ICs is *interaction-free*, we can compute consistent answers to projection free conjunctive queries by using the deterministic set of program $\Pi(D, IC, \mathcal{Q})$. This set is computed by DLV system in polynomial time.

Chapter 8

A Repair Semantics for Multidimensional Databases

8.1 Introduction

In this chapter we present a semantic framework for CQA in Multidimensional Databases (MDBs) [50, 51]. Specifically, we focus in Multidimensional Data Warehouses (MDWs), which are data repositories that integrate and materialize data from different sources and also keep historical data [24]. They can be queried by *OLAP* (On-Line Analytical Processing) systems, which in particular, require aggregation of data stored in the data warehouse [24].

The MDWs consist mainly of dimensions and facts. Dimensions reflect the way in which the data is organized. Some typical dimensions are time, location, customers, products, etc. The facts correspond to quantitative data related with the dimensions. For example, facts related with sales may be associated to the dimensions time, product, and location; and should be understood as the sales of products at the locations at certain periods of time.

Data warehouses can be modelled and implemented by using a relational (ROLAP) or a multidimensional (MOLAP) approach. In the former, the data is stored in relational databases, and special access methods are develop in order to efficiently implement the aggregation of data, which is the most common task in data warehouses. In the latter, the multidimensional data is stored in special data structures, and aggregate operations are implemented over these special data structures [24].

The multidimensional approach is better than the relational one to support data aggregation, because aggregations can be computed in a straightforward way from the multidimensional structure. We base our work on the *multidimensional model* proposed in [48, 49], where dimensions are modelled by hierarchy schemas together with a set of constraints, while the facts are represented by tables that refer to the dimensions. Here, we only consider basic dimension schemas, called *strictly homogeneous dimension schemas* (cf. section 8.2), but not other complex schemas such as the *heterogeneous dimension schemas* [48, 49].

Usually, dimensions are considered the static part of the MDWs, whereas the facts are considered to be the dynamic part, in the sense that the update operations affect mainly the fact tables. In [50, 51] the need for updates on dimensions is analyzed. The authors argue that dimensions have to be adapted to changes in data sources or to the business structure. They define a set of update operators for homogeneous dimension schemas and instances.

In the presence of such update operations, MDWs may become inconsistent wrt dimension constraints. We are interested in studying the effects of violations of a specific class of dimension constraints, the so-called *partitioning constraints* in homogeneous dimension instances (from now on, homogeneous instances). These constraints are fundamental for enforcing navigability properties in dimension schemas. One of the effects that we will analyze in detail is how, the violation of constraints, affects the *summarizability* property (*SUMM*) of the MDWs, which is the capability of correctly computing queries (*cube views*) using other pre-computed aggregate views. We will concentrate on aggregate queries with *group-by* statements, that is, queries that perform grouping of attributes and return a value for each group (cf. Chapter 6).

We also intend to retrieve consistent answers to queries, even when the MDWs are inconsistent. Therefore, a characterization of such answers becomes necessary. In order to do this, we use the concept of *repair* of MDWs that are inconsistent wrt the dimension constraints. In this respect, we show that the previous notions of repairs, e.g. the relational notion given in [2], are not suitable for MDWs. In consequence, we give a new definition of repair for MDWs subject to sets of partitioning constraints.

We define repairs of dimension instances wrt partitioning constraints by introducing minimal changes over the original inconsistent dimension instances. In order to achieve this, and given that we are considering hierarchical representations with multiple levels, we explore the notion of prioritized minimization (as given in [62]).

The rest of the chapter is organized as follows: Section 8.2 reviews the multidimensional model, including partitioning constraints. In Section 8.3 we discuss the necessity of defining a new notion of repair for MDBs. In Section 8.4 the notions of dimension instance repair and consistent answer to aggregate queries are presented. This version of repair is used as an auxiliary element to compute consistent answers to aggregate queries. Section 8.5 finalizes this chapter.

8.2 The Multidimensional Model

A hierarchy schema is a directed acyclic graph $\mathcal{H} = (C, \nearrow)$, where C is a set of *categories*, and $\nearrow$ is a child/parent relation between categories (edges in the graph), i.e. for a pair of categories $C_1, C_2 \in C$, we write $C_1 \nearrow C_2$ to denote that (C_1, C_2) is an edge in $\mathcal{H}$. $\nearrow^*$ is the transitive and reflexive closure of $\nearrow$. For simplicity, categories do not have any attributes, and for technical reasons, there is a distinguished *top category* named *All*, whose only element is $\{all\}$, which is reachable from all other categories via $\nearrow^*$. The category at the lowest level is named the *bottom* category.[1]

Example 8.1 The National Parks' hierarchy schema is defined by:

- A set of categories $C = \{$*Park, Type, Location, Country, All*$\}$,

[1]We restrict ourselves to dimension schemas with only one *bottom* category. But, they may have more than one *bottom* category.

- The child/parent relation $\nearrow$ consisting of the edges $\{$(*Park, Type*), (*Park, Location*), (*Type, Country*), (*Location, Country*),(*Country, All*)$\}$; and
- $\nearrow^{*}=\nearrow \cup$ $\{$(*Park,Park*), (*Type, Type*), (*Park, Country*), ...$\}$

This hierarchy schema is shown in Figure 8.1.

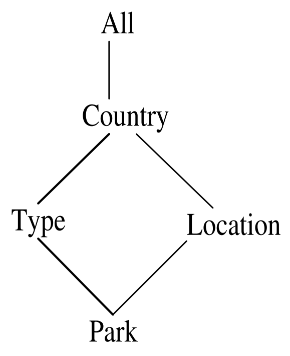


Figure 8.1: National Parks' Hierarchy Schema

We can see that the bottom category is *Park*, and that *Type* and *Location* are direct ancestors of *Park*, being *Country* and *All* indirect ancestors. □

The hierarchy schema has a domain $\mathcal{U}$ that can be infinite. The categories and their elements contain values from $\mathcal{U}$, and for two names n_1, n_2 it holds $n_1 \neq n_2$ (unique name assumption). An instance over a hierarchy schema can be represented as a first-order logic structure of the form:

$$\mathcal{D} = \langle \mathcal{U}, C_1^{\mathcal{U}}, ..., C_m^{\mathcal{U}}, c_1^{\mathcal{U}}, \ldots, c_n^{\mathcal{U}}, C_{bottom}^{\mathcal{U}}, All^{\mathcal{U}}, all^{\mathcal{U}}, A^{\mathcal{U}}, <^{\mathcal{U}}, <^{*\mathcal{U}} \rangle, \tag{8.1}$$

where $\mathcal{U}$ is the Herbrand domain of $\mathcal{H}$ [64], whose elements must be interpreted with their own values. $C_1^{\mathcal{U}}, ..., C_m^{\mathcal{U}}, C_{bottom}^{\mathcal{U}}, All^{\mathcal{U}} \subseteq \mathcal{U}$, are unary predicates that represent categories, $C_1^{\mathcal{U}} = C_{bottom}^{\mathcal{U}}$ is the bottom category, $C_m^{\mathcal{U}} = All^{\mathcal{U}} = \{all^{\mathcal{U}}\}$ is the top category, and $c_1^{\mathcal{U}}, \ldots, c_n^{\mathcal{U}}$ are names for elements of categories. $A^{\mathcal{U}} \subseteq \{p_{Ci,Cj} \mid i, j =$

$1, ..., m\}$, where each $p_{Ci,Cj}$ represents an edge between the categories $C_i^{\mathcal{U}}$ and $C_j^{\mathcal{U}}$ on the hierarchy schema. There is a child/parent relation between elements of categories, which is represented by $<^{\mathcal{U}} \subseteq \mathcal{U} \times \mathcal{U}$. $<^{*^{\mathcal{U}}} \subseteq \mathcal{U} \times \mathcal{U}$ is the reflexive and transitive closure of $<^{\mathcal{U}}$. In this sense, $<^{*^{\mathcal{U}}}$ can be seen as an interpreted relation name, which has a fixed interpretation depending on the interpretation of $<^{\mathcal{U}}$. A dimension instance D indicates relationships between those categories that must be connected in the hierarchy schema. That $p_{Ci,Cj} \in A^{\mathcal{U}}$ indicates that $C_i \nearrow C_j$.

Example 8.2 (example 8.1 cont.) $D = \langle \mathcal{U}$, $\mathit{Park}(\cdot)^{\mathcal{U}}$, $\mathit{Type}(\cdot)^{\mathcal{U}}$, $\mathit{Location}(\cdot)^{\mathcal{U}}$, $\mathit{All}(\cdot)^{\mathcal{U}}$, $\mathit{Country}(\cdot)^{\mathcal{U}}$, $\mathit{all}^{\mathcal{U}}$, $A^{\mathcal{U}}$, $<^{\mathcal{U}}$, $<^{*^{\mathcal{U}}} \rangle$[2] is an instance for the hierarchy schema in Figure 8.1, where $\mathcal{U}$ is composed by names for categories, parks, types, locations, countries, i.e. $\mathcal{U} = \{$ *Park*, *Type*, *Location*, *Country*, *Banff, Jasper, Crater Lake*, *P, S*, *Alberta, Oregon*, *Canada, US*, *All, all*$\}$, and:

- $\mathit{Park}^{\mathcal{U}} = \{\mathit{Banff}, \mathit{Jasper}, \mathit{CraterLake}\}$, $\mathit{Type}^{\mathcal{U}} = \{P, S\}$,[3]
  $\mathit{Location}^{\mathcal{U}} = \{\mathit{Alberta}, \mathit{Oregon}\}$, $\mathit{Country}^{\mathcal{U}} = \{\mathit{Canada}, \mathit{US}\}$, $\mathit{All}^{\mathcal{U}} = \{\mathit{all}^{\mathcal{U}}\}$.

- $A^{\mathcal{U}} = \{p_{\mathit{Park},\mathit{Type}}, p_{\mathit{Park},\mathit{Location}}, p_{\mathit{Type},\mathit{Country}}, p_{\mathit{Location},\mathit{Country}}, p_{\mathit{Country},\mathit{All}}\}$.

- $<^{\mathcal{U}} = \{$(*Banff, P*), (*Banff, Alberta*), (*Jasper, P*), (*Jasper, Alberta*),
  (*Crater Lake, S*), (*Crater Lake, Oregon*), (*P, Canada*), (*S, US*),
  (*Alberta, Canada*), (*Oregon, US*), (*Canada, All*), (*US, All*)$\}$.

- $<^{*^{\mathcal{U}}} = <^{\mathcal{U}} \cup \{$(*Banff, Banff*), (*Banff,Canada*)... $\}$.

Figure 8.2 is the dimension instance D.

$\square$

[2] $P(\cdot)$ for any category P indicates that the arity of P is one.
[3] P denotes Province and S denotes State.

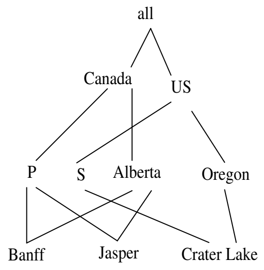


Figure 8.2: National Parks' Dimension Instance

The properties of hierarchy schemas and their instances can be expressed with a first-order language. This, by using the symbols of $\mathcal{H}$ plus the symbols $\nearrow^*$, $<$, $<^*$, etc.

A *roll-up function* is a relation based on $<^*$ between elements of two fixed categories:

$$\mathcal{R}_{C_i}^{C_j}(D) := \{(x,y) \mid C_i^{\mathcal{U}}(x) \wedge C_j^{\mathcal{U}}(y) \wedge x <^* y\} \tag{8.2}$$

We can see that those functions are parameterized by a pair of categories and depend on the instance D at hand. These functions are fundamental for computing aggregation of data.

Dimension instances must satisfy a set of constraints [48]. The *partitioning* constraint is one of them, and for a pair of categories C_i and C_j is defined by the following $\mathcal{L}$-sentence:

$$\forall xyz(C_i(x) \wedge C_j(y) \wedge C_j(z) \wedge x <^* y \wedge x <^* z \rightarrow y = z). \tag{8.3}$$

It enforces that roll-up functions are functional, and so they allow for the correct computation of aggregations.

In data warehousing, aggregation of data or *cube views* can be performed at different granularities, i.e. set of categories from the dimension schemas. The notion of summarizability ($SUMM$) in dimension schemas [48, 49] refers to the capability of correctly compute any *cube view* defined at a category C_i from a pre-computed aggregate view defined at category C_j by using the roll-up functions between categories C_i y C_j.

Definition 8.1 [48, 49] A category C_i is summarizable from a category C_j in a dimension instance D of $\mathcal{H}$ iff (a) $C_j \nearrow^* C_i$, and (b) $R^{C_i}_{Cbottom}(D) = R^{C_j}_{Cbottom}(D) \bowtie R^{C_i}_{C_j}(D)$, where $\bowtie$ is the *join* operator [1]. □

Example 8.3 (example 8.2 cont.) The partitioning constraint is: $\forall xyz(Park(x) \wedge Country(y) \wedge Country(z) \wedge x <^* y \wedge x <^* z \rightarrow y = z)$, which establishes that every element of the category *Park* has to be associated with a unique element in the category *Country*. The roll-up function between categories *Park* and *Country* is $R^{Country}_{Park}(D) = \{(Banff, Canada), (Jasper, Canada), (CraterLake, US)\}$, which by the $SUMM$ property, can be computed by using the intermediate category *Type* and the roll-up functions $R^{Type}_{Park}(D) = \{(Banff, P), (Jasper, P), (CraterLake, S)\}$ and $R^{Country}_{Type}(D) = \{ (P, Canada), (S, US)\}$, since $R^{Country}_{Park}(D) = R^{Type}_{Park}(D) \bowtie R^{Country}_{Type}(D)$. □

There are other constraints to model hierarchy schemas [48, 49]. Those are used to specify paths and the existence of distinguished elements in the categories. We call to them *specific dimensions constraints*. In particular, the dimension constraints that establish conditions on paths between categories are called *path constraints*. Those that specify that paths are mandatory are called *into constraints*. The dimension constraints that specify the existence of values in categories are called *equality constraints*.

Example 8.4 For the hierarchy schema in Figure 8.1, the following $\mathcal{L}$-sentences are dimension constraints:

(a) $\forall x(Park(x) \rightarrow \exists y(Type(y) \wedge x < y))$. This is an *into constraint* that establishes that every element in $Park$ is associated (rolls-up) with an element in $Type$.

(b) $\exists x(Country(x) \wedge x = Canada)$. This is an *equality constraint* that establishes that there exists an element named $Canada$ in category $Country$.

(c) $\forall xy(Park(x) \wedge Type(y) \wedge y = P \wedge x < y \rightarrow \exists z(Country(z) \wedge x <^* z \wedge z = Canada))$, which asserts that for every element x in $Park$, if x rolls-up to P in $Type$, then x is associated with $Canada$ in $Country$. This is a combination of a path and an equality constraint. □

An *homogeneous dimension schema* is a schema modelled by *into constraints*. Therefore, all the paths on it are mandatory. In homogeneous schemas, roll-up functions are expected to be total between elements of categories. A schema is called *strictly homogeneous* when it has one bottom category, e.g. the hierarchy schema in Figure 8.1.

In data warehouses *cube views*, i.e. aggregate queries, are computed from the dimensions instances through the roll-up functions and the fact tables. The roll-up functions are treated as relational tables. For instance, the roll-up function R_{Park}^{Type} can be seen as a relational table with schema $R(Park, Type)$.

We will concentrate on aggregate queries with *group-by* statements. In MDWs an aggregate query is of the form:

$$\begin{array}{l} \texttt{SELECT}\ A_j, \ldots A_n, f(A) \\ \texttt{FROM}\ T,\ R_i, \ldots R_m\ \texttt{WHERE conditions} \\ \texttt{GROUP BY}\ A_j, \ldots A_n \end{array} \tag{8.4}$$

where $A_j, \ldots A_n$ are attributes of the fact table T or of the roll-up functions $R_i, \ldots R_m$ (treated as tables), and `f` is one of `min(A)`, `max(A)`, `count(A)`, `sum(A)`, `avg(A)`, applied to attribute A with $A \cap \{A_j, \ldots A_n\} = \emptyset$.

Example 8.5 (example 8.3 cont.) Consider the facts table *Sales* below storing sales for national parks, and the roll-up function $R_{Park}^{Type}(D) = \{(Banff, P), (Jasper, P), (CraterLake, S)\}$, which can be seen as the relational table $R(Park, Type)$:

Sales	Park	Amount
	Banff	5000
	Jasper	5000
	Crater Lake	10000

R	Park	Type
	Banff	P
	Jasper	P
	Crater Lake	S

For the aggregate query:

```
SELECT R.Type, SUM(S.Amount)
FROM Sales S, R
WHERE R.Park = S.Park
GROUP BY R.Type
```

The answers are $(P, 10000)$, $(S, 10000)$. □

8.3 The Need for MDWs Repairs and Consistent Answers

In general, MDWs are conceived as collections of materialized views whose main sources are operational databases. As a consequence, much effort has been centered around keeping consistency between the sources and the MDWs [40, 41, 72, 74]. To the best of our knowledge, the first work related to consistency in dimension schemas in the sense of satisfiability of partitioning constraints [48] is presented in [56]. The authors argue that a dimension schema is consistent if their instances satisfy the partitioning constraints. The notion of consistency the authors present is used to guide the update operations on dimension schemas in such a way that the partitioning constraints are satisfied.

However, there has been no work, so far, that tackles the problem of already having an inconsistent dimension instance wrt a specific class of constraints, but still being able to provide consistent answers, in a sense similar to the notion of consistent answer introduced in [2, 5, 7] for relational databases. In this regard, the work presented here is the first attempt to handle the problem of consistent query answering in MDWs.

Here, we analyze the effect, over query answering, of having inconsistent dimension instances wrt partitioning constraints. We illustrate this in the example below.

Example 8.6 (example 8.5 cont.) The dimension instance in Figure 8.3 is inconsistent wrt the partitioning constraint: $\forall xyz(Park(x) \wedge Country(y) \wedge Country(z) \wedge x <^* y \wedge x <^* z \rightarrow y = z)$. This is because the element *Crater Lake* rolls-up to *Canada* via type *P*, and also to *US* via location *Oregon*. As a consequence, the roll-up function $R_{Park}^{Country} = \{(\mathit{Banff}, \mathit{Canada}), (\mathit{Jasper}, \mathit{Canada}), \underline{(\mathit{CraterLake}, \mathit{Canada})}, \underline{(\mathit{CraterLake}, \mathit{US})}\}$ is not functional.

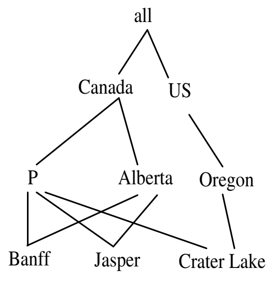


Figure 8.3: Inconsistent Dimension Instance

Suppose the facts table *Sales* in Example 8.5, the roll-up function $R_{Park}^{Country}$ represented by the relational table $R(Park, Country)$ below, and the aggregate query:

```
SELECT R.Country, SUM(S.Amount)
FROM Sales S, R
WHERE R.Park = S.Park
GROUP BY R.Country
```

R	Park	Country
	Banff	Canada
	Jasper	Canada
	Crater Lake	Canada
	Crater Lake	US

The answers to the query are $(Canada, 20000)$, $(US, 10000)$. □

Clearly, this result presents an anomaly, the sales of the park *Crater Lake* are added twice, as sales of *Canada* and also as sales of *US*.

Now, let us explore how that violation affects the summarizability property.

Example 8.7 (example 8.6 cont.) Consider the roll-up functions $R_{Park}^{Type} = \{(Banff, P)$, $(Jasper, P)$, $(CraterLake, P)\}$, and $R_{Park}^{Location} = \{(Banff, Alberta)$, $(Jasper, Alberta)$, $(CraterLake, Oregon)\}$, which are computed on the inconsistent dimension instance in Figure 8.3, and that represented as tables become, respectively:

R_1	Park	Type
	Banff	P
	Jasper	P
	Crater Lake	P

R_2	Park	Location
	Banff	Alberta
	Jasper	Alberta
	Crater Lake	Oregon

Consider also the materialized views *Sales-Type* and *Sales-Loc*, which are defined as follows:

```
SELECT R1.Type, SUM(S.Amount)
FROM Sales S, R1
WHERE R1.Park = S.Park
GROUP BY R1.Type
```

```
SELECT R2.Location, SUM(S.Amount)
FROM Sales S, R2
WHERE R2.Park = S.Park
GROUP BY R2.Location
```

Sales-Type	Type	Amount
	P	20000

Sales-Loc	Location	Amount
	Alberta	10000
	Oregon	10000

The roll-up functions $R_{Type}^{Country} = \{(P,\ Canada)\}$, and $R_{Location}^{Country} = \{(Alberta,\ Canada),$ $(Oregon,\ US)\}$, seen as the relational tables:

R_3	Type	Country
	P	*Canada*

R_4	Location	Country
	Alberta	*Canada*
	Oregon	*US*

For the queries $\mathcal{Q}$ and $\mathcal{Q}$', respectively:

```
SELECT $R_3$.Country, SUM(S.Amount)
FROM Sales-Type S, $R_3$
WHERE $R_3$.Type = S.Type
GROUP BY $R_3$.Country,
```

```
SELECT $R_4$.Country, SUM(S.Amount)
FROM Sales-Loc S, $R_4$
WHERE $R_4$.Location = S.Location
GROUP BY $R_4$.Country
```

The answer to $\mathcal{Q}$ is $(Canada, 20000)$, and the answers to $\mathcal{Q}$' are $(Canada,\ 10000)$, $(US,\ 10000)$. Nevertheless, by the summarizability property, the answers must be the same. $\square$

It has been shown that for strictly homogeneous dimensions that satisfy their partitioning constraints, the summarizability property is guarantee [48]. In fact, in those schemas a category C_i is summarizable from any of their child categories, i.e. from any C_j such that $C_j \nearrow^* C_i$.

This is important because we can verify summarizability by testing satisfiability of partitioning constraints. This test could be easily performed by using views. In this way, we could also identify the elements participating in violations of partitioning constraints and use that information to fix dimension instances. We illustrate this in the example below.

Example 8.8 (example 8.6 cont.) To check if there are elements of category *Park* that roll-up to different elements in category *Country*, we can define the following view *Check* which captures the tuplas from R, which is the relational representation of the roll-up function $R_{Park}^{Country}$:

```
CREATE VIEW Check AS
SELECT *
FROM R
WHERE EXISTS (SELECT *
              FROM R R2
              WHERE R2.Park = R.Park AND R2.Country <> R.Country)
```

If the view *Check* is evaluated on relation R it returns tuples (*Crater Lake, Canada*), (*Crater Lake, US*). Therefore, we can conclude that the dimension instance violates the partitioning constraint, since the roll-up between elements of categories *Park* and *Country* is not functional. □

The notion of database repair for relational databases does not capture the minimality required by the natural process of repairing MDWs. This, when the MDW is represented as an instance of a non-normalized relational database (the ROLAP approach). We illustrate this with an example.

Example 8.9 (example 8.6 cont.) Figure 8.4 is a star schema [24], that is a non-normalized relational representation for the National Parks' dimension, where the primary key for each table appears underlined. The partitioning constraint can be enforced by the following first-order integrity constraint $IC\colon \forall xyzwv(\mathit{Park}(x,y,z) \wedge \mathit{Type}(y,w) \wedge \mathit{Country}(w) \wedge \mathit{Location}(z,v) \wedge \mathit{Country}(v) \rightarrow w = v)$.

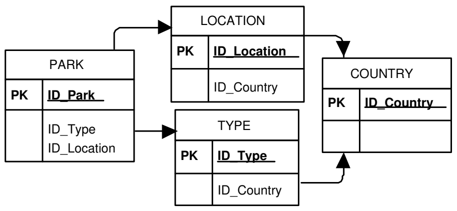


Figure 8.4: Star Schema for National Parks' Dimension

The dimension instance D below is the relational representation of the inconsistent dimension instance in Figure 8.3.

Park	ID_Park	ID_Type	ID_Location
	Banff	*P*	*Alberta*
	Jasper	*P*	*Alberta*
	CraterLake	*P*	*Oregon*

Country	ID_Country
	Canada
	US

Location	ID_Location	ID_Country
	Alberta	*Canada*
	Oregon	*US*

Type	ID_Type	ID_Country
	P	*Canada*

In a relational sense, consistency can be restored [2] by deleting tuple $Park(Crater\ Lake, P, Oregon)$. This relational tuple represents the edges $(Crater\ Lake, P)$ and $(Crater\ Lake, Oregon)$ in the multidimensional schema. Therefore, if we apply this change in the multidimensional representation in Figure 8.3, we have to delete both edges . However, we will see in Section 8.4 this is not a minimal change in a multidimensional sense, since consistency will be reestablished by eliminating just one of them.

Notice that if we use a normalized snowflake schema [24] to model the dimension $Park$, we will obtain the relations $Park\_Type(ID\_Park, ID\_Type)$ and $Park\_Location$

(*ID_Park*, *ID_Location*). In this new schema, consistency will be restored by deleting tuple *Park_Type*(*Crater Lake*, *P*), or tuple *Park_Location*(*Crater Lake*, *Oregon*), which corresponds to the elimination of edges (*Crater Lake*, *P*) and (*Crater Lake*, *Oregon*) in Figure 8.3, respectively. Nevertheless, we want to be able of working directly in the multidimensional dimension instance, which usually, by query optimization purposes, cannot be represented by a set of normalized relational tables [24]. □

The idea of changing the relation schema before repairing inconsistencies is presented in [77]. In this paper, a project-join dependency is applied to relations prior to repairing them wrt FDs by tuple deletion. In this way, more meaningful repairs wrt FDs are obtained.

The problem in the non-normalized relational model is that a tuple in a dimension table may represent many pairs of edges in the multidimensional representation (in Example 8.9 the tuple represents two pairs). In that sense, the relational model does not allow us to work on a granularity lower than a tuple.

Our definition of repair (cf. Section 8.4) captures exactly the minimality of changes desired for MDWs. We achieve this by first identifying the edges on dimension instances that involve elements of categories that participate in violations of partitioning constraints. Then, since the roll-up functions capture the edges in a dimension instance, we identify the roll-up functions that contain pairs of inconsistent elements, and we eliminate pair of elements from them to restore consistency. Intuitively, the repairs are those that recover consistency by doing prioritized minimal changes over the dimension instance (inspired by the notion of prioritized minimization given in [62]).

The importance of the summarizability property of MDWs has been analyzed in

[53, 65]. In [65] a particular class of heterogeneous hierarchies is transformed into homogeneous hierarchies to support summarizability. This is achieved by inserting null values, merging other values, and introducing new categories when partitioning constraints are violated. Although these operations, which allow us to get summarizability, could be used for repairing inconsistent MDWs, they do not produce minimal repairs. In addition, we believe that the fusion of values may produce undesired changes in the semantics of the dimension instances.

8.4 Repairs and Consistent Answers in MDWs

In this section we give the notion of repair for multidimensional dimension instances. The notion of repair is used as an auxiliary concept to compute consistent answers to aggregate queries.

8.4.1 Repairs of Dimension Instances

Partitioning constraints can be seen as functional dependencies in relational databases. The general way to repair inconsistent databases wrt FDs is by deleting the tuples participating in the violations [5]. However, in dimension instances, there are no tuples in the sense of relational databases, but there exist dimension tuples [48], so we could consider as tuples the pairs in the roll-up functions (treated as tables).

Example 8.10 (example 8.7 cont.) The roll-up function $R_{Park}^{Location} = \{(Banff, Alberta),$ $(Jasper, Alberta), (CraterLake, Oregon)\}$ can be seen as the relational table:

R	Park	Type
	Banff	P
	Jasper	P
	Crater Lake	P

Here, the dimensional tuples are $(\mathit{Banff}, P), (\mathit{Jasper}, P)$, and $(\mathit{CraterLake}, P)$. □

We can see that hierarchical dimension instances determine a set of roll-up functions. This is, there are roll-up functions for every pair of categories that are connected in the dimension schema. Moreover, the roll-up functions may contain elements that participate in violations of partitioning constraints. Then, we should identify from which roll-up functions pairs are to be deleted in order to get a good repair. Inspired by the notion of prioritized minimization given in [62], we propose to minimize changes, but assigning higher priority to lower categories. In this manner, we ensure that the edges between categories that are involve directly in inconsistencies will be deleted. For this purpose, we define levels of categories on a dimension instance, and associate a set of roll-up functions to each level.

Definition 8.2 Given a dimension instance D of the form (8.1) with dimension schema $\mathcal{H}$, and maximum distance n between the categories in $\mathcal{H}$. A level L_i, with $0 \leq i \leq n$, is a set of elements belonging to categories with distance i to the bottom category. For each level L_i, and categories C_j, C_k in $\mathcal{H}$, there exists a set $R_i \subseteq <^*$ defined by $R_i := \{(a, b) \mid a < b \wedge C_j(a) \in L_i \wedge C_k(b) \in L_{i+1}\}$.[4] □

Example 8.11 For the dimension instance D in Figure 8.3, we have:

- $L_0 = \{\mathit{Banff}, \mathit{Jasper}, \mathit{CraterLake}\}$, $R_0 = \{(\mathit{Banff}, P), (\mathit{Banff}, \mathit{Alberta}), (\mathit{Jasper}, P), (\mathit{Jasper}, \mathit{Alberta}), (\mathit{CraterLake}, P), (\mathit{CraterLake}, \mathit{Oregon})\}$.

- $L_1 = \{P, \mathit{Alberta}, \mathit{Oregon}\}$, $R_1 = \{(P, \mathit{Canada}), (\mathit{Alberta}, \mathit{Canada}), (\mathit{Oregon}, \mathit{US})\}$.

- $L_2 = \{\mathit{Canada}, \mathit{US}\}$, $R_2 = \{(\mathit{Canada}, \mathit{all}), (\mathit{US}, \mathit{all})\}$.

[4]Where $<$ is the child/parent relation between elements of categories.

- $L_3 = \{all\}$, $R_3 = \emptyset$. $\square$

In order to define repairs of dimension instances we need to introduce some concepts.

Definition 8.3 Let D, D', D'' be dimension instances over the same dimension schema and domain. The distance, between D' and D'' at level L_i, with $0 \leq i \leq n$ is defined by $\Delta_i(D', D'') := \{(a, b) \mid (a, b) \in R_i$ of D', and $(a, b) \notin R_i$ of D'', or $(a, b) \in R_i$ of D'', and $(a, b) \notin R_i$ of $D'\}$, where R_i is the set of roll-up functions at level L_i. In other words, $\Delta_i(D', D'')$ contains all the pairs (a, b) that do not appear in both dimension instances D', D'' at certain level L_i.

There is a parcial order between dimension instances:

(a) It holds $D' \leq_{D,i} D''$ iff for every dimension tuple $(a, b) \in \Delta_i(D, D')$, it holds that $(a, b) \in \Delta_i(D, D'')$.

(b) It holds $D' \leq_D D''$ iff there exists an i such that $\Delta_k(D, D') = \Delta_k(D, D'')$, for $k < i$, and $D' \leq_{D,i} D''$. $\square$

A dimension instance repair wrt a set of partitioning constraints *MPC* is defined as follows:

Definition 8.4 Given a dimension instance D, and a set of partitioning constraints *MPC*, a repair of D wrt *MPC* is a dimension instance D', over the same dimension schema H, and domain $\mathcal{U}$, such that $D' \models \mathit{MPC}$, and D' is $\leq_D$-minimal in the class of dimension instances that satisfy *MPC*.

The set of repairs of D is denoted by $\mathit{Repairs}_{\mathit{MPC}}(D)$. $\square$

Example 8.12 (example 8.9 cont.) Figure 8.5 shows the dimension instance repairs for the dimension instance D in Figure 8.3.

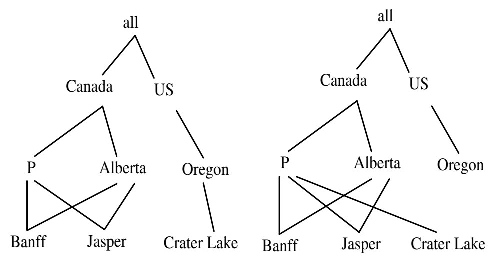


Figure 8.5: Dimension Instance Repairs D' and D''

Here, $\Delta_0(D, D') = \{(\mathit{CraterLake}, P)\}$, and $\Delta_0(D, D'') = \{(\mathit{CraterLake}, \mathit{Oregon})\}$.

The dimension instance D''' in Figure 8.6 is not a repair.

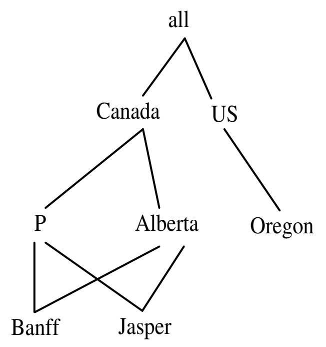


Figure 8.6: Non-Minimal Dimension Instance

This is because $\Delta_0(D, D''') = \{(\mathit{CraterLake}, P), (\mathit{CraterLake}, \mathit{Oregon})\}$ and it holds that $D' \leq_{D,0} D'''$, and $D'' \leq_{D,0} D'''$. □

The repairs of dimension instances satisfy the partitioning constraints, however they may not be homogeneous instances anymore.

Proposition 8.1 Let D, D' be dimension instances over the same dimension schema and domain. If D' is a repair of the homogeneous instance D, then D' is not homogeneous.

Proof: In homogeneous schemas the roll-up functions between elements of categories are expected to be total. This is, for every element x of a category C_i, if $C_i \nearrow^* C_j$

in a hierarchy schema H, then there exists an element y in C_j such that $x <^* y$ in a dimension instance D of H. It is easy to see that if D' is a repair of D, there exists a pair of categories C_j, C_k in H such that $C_j \nearrow^* C_k$, for which the roll-up function $R_{C_j}^{C_k}$ in D' contains a subset of the pairs in the roll-up function $R_{C_j}^{C_k}$ in D. Therefore, in D', there is an element y in C_j for which there is no element z in C_k such that $y <^* z$. Hence, the roll-up function $R_{C_j}^{C_k}$ is not total, and as a consequence, D' is not homogeneous. $\square$

This happens because consistency is recovered by deleting edges in dimension instances, i.e. pairs from roll-up functions. As an illustration, in the first repair of Figure 8.5 the roll-up function between categories *Park* and *Type* is not total, because element *Crater Lake* is not related with an element of *Type*. Moreover, in the second repair, there is no relation between element *Crater Lake* and an element of *Location*. Therefore the roll-up function between *Park* and *Location* is not total either. As a consequence, the summarizability property may not be satisfied in the repairs. However, we could obtain total functions and also summarizability by introducing "dummy" elements in some categories in the repairs, as in [65]. It becomes interesting to analyze the possible advantages of implementing the idea of using "dummy" elements, in the query answering process for MDWs.

Also, we could use the method proposed in [65] to repair MDWs and compute consistent answers.

8.4.2 Consistent Answers

We adopt the notion of consistent answers wrt FDs to aggregate queries with *group by* statements presented in Chapter 6. In multidimensional databases aggregate queries are computed from roll-up functions, and fact tables, both represented as relational tables.

A tuple of the form $\langle t_1, \ldots, t_m, [a, b]\rangle$ is a consistent answer to an aggregate query with *group-by* statements of the form (8.4), wrt a set MPC of partitioning constraints, if first $\langle t_1, \ldots, t_m\rangle$ is a consistent answer in the usual sense, i.e. it is true in every repair of the dimension instance, and second, if for every repair there exists an aggregate value for the group of attributes values in $t_1, \ldots, t_m$, such that it falls in the minimal numerical interval $[a, b]$.

Example 8.13 (example 8.6 and 8.12 cont.) The roll-up function $R_{Park}^{Country}$ in dimension instances D' and D'' (cf. Figure 8.5) are (represented as relational tables), respectively:

R	Park	Country
	Banff	*Canada*
	Jasper	*Canada*
	Crater Lake	*US*

R	Park	Country
	Banff	*Canada*
	Jasper	*Canada*
	Crater Lake	*Canada*

For the aggregate query $\mathcal{Q}$:

```
SELECT R.Country, SUM(S.Amount)
FROM Sales S, R
WHERE R.Park = S.Park
GROUP BY R.Country
```

The answers from repair D' are $\{(\textit{Canada}, 10000), (\textit{US}, 10000)\}$, and $\{(\textit{Canada}, 20000)\}$ from D''. Therefore, the only consistent answer to the query is $(\textit{Canada}, [10000, 20000])$, as expected. □

8.5 Summary

In this chapter we presented a repair semantics for MDBs, specifically we concentrated our work on multidimensional data warehouses (MDWs). It was shown that the relational notion of repair [2, 5, 7] does not capture the minimality required by the natural process of repairing MDWs. This, when the MDW is represented as an instance of a non-normalized relational database. However, the notion of database repair we presented captures the minimality required in MDBs.

It is relevant to analyze the idea of translating dimension instances into sets of normalized relational tables before repairing them. In this way, we could apply the notion of relational database repair of [2]. This idea is presented in [77], where a project-join dependency is applied to a relation prior to repairing wrt FDs by tuple deletion. In this way, they obtain more meaningful repairs wrt FDs. Moreover, it is important to analyze the gains and costs, in terms of execution of queries, of doing such transformation. This is important because, a non-normalized representation of dimension instances, allows a better execution time of queries [24].

The repair definition of dimension instances can be improved by using knowledge from equality atoms constraints [48], which impose the existence of certain distinguished elements in the categories. We could require that repairs satisfy those constraints to get more meaningful repairs.

For future research, we leave the development of a methodology for computing repairs (if necessary, because this should be avoided whenever possible due to its complexity), and consistent answers.

This is a preliminary study that we will extend to heterogeneous dimension schemas [48], which are modelled by other kind of dimension constraints. Moreover, we are interested in analyzing consistent query answering wrt aggregation constraints [73]. This issue is still open.

Chapter 9

The Consistency Extractor System

9.1 Introduction

In this chapter we describe the "Consistency Extractor System", from now on just named the *ConsEx* system. *ConsEx* computes consistent answers to FO queries posed to inconsistent databases that may contain null values. It implements the logic programs of Definition 4.2, and adopts the semantics of ICs satisfaction defined in [16], and presented in Chapter 2. The system considers UICs, RICs and NNCs of the forms (2.3), (2.4), and (2.6), respectively.

ConsEx implements two methods to compute consistent answers, which can be selected by the user before the evaluation of queries. The first one is the MS methodology defined in Chapter 5. MS simulates a top-down [23] -and then query directed-evaluation of the query through bottom-up propagation [23]. This technique produces a new program that contains a subset of the original rules, those that are relevant to evaluate the query, along with a set of new, "magic", rules.

The second methodology takes advantage of the relevant predicates to compute query answers in the generation of repair programs (cf. Definition 5.6). In this way, *ConsEx* generates repair programs for a subset of the database predicates, those that will be involved in the computation of the query. Thus, query answers are computed by running the new, smaller repair program together with the query program.

We show that the methods implemented in *ConsEx* compute consistent answers to queries faster than the straightforward evaluation of programs (cf. Section 9.4).

Another goal is to take advantage of the interaction between logic programming environments and database management systems (DBMS), by exploiting all the capabilities of the DBMS such as storing and indexing; and taking advantage of the robustness of logical reasoning as provided by the DLV system [61]. For instance, bringing the whole database into DLV, to compute consistent answers, is inefficient. Therefore, we reduce as much as possible the interaction between the DBMS, where the data resides, and the logic programming environments, where the repair programs are evaluated.

ConsEx is the first attempt to implement optimized methods for CQA based on logic programs for relational databases, considering a broad range of integrity constraints, like UICs, RICs and NNCs.

As far as we know, the *INFOMIX* system [58] implements logic programs for CQA, in the context of data integration [54, 57] under the *GAV* approach [54]. *INFOMIX* implements the logic approach presented in [34], which works for UICs, but it does not support other ICs like RICs. In *INFOMIX* queries are restricted to the union of conjunctive queries, when the ICs involve inclusion dependencies, and in absence of them, queries can be recursive Datalog queries with aggregates and stratified negation.

The rest of the chapter is organized as follows: Section 9.2 presents the architecture of *ConsEx*. Section 9.3 describes the graphical user interface of *ConsEx*. Section 9.4 describes the experimental evaluation and the results. Section 9.5 finalizes this chapter.

9.2 ConsEx Architecture

The input to *ConsEx* consists of the database parameters, like database name, ip (for remote connections), user and password; FO queries; ICs; and the options of the methodology selected by the user to evaluate the query, i.e. the MS technique or the methodology that generates programs for the relevant predicates to compute queries. From now on, the latter methodology will be referenced as the RP (for relevant predicates) methodology.

We will use the example below (which is a variation of Example 5.1) to describe the architecture of *ConsEx*.

Example 9.1 Consider the database schema $\Sigma = \{S(ID), Q(ID), R(ID), T(ID), W(ID)\}$, database instance $D = \{S(a), T(a), S(b), Q(b), R(b)\}$, a set IC with $\forall x (S(x) \rightarrow Q(x))$, $\forall x(Q(x) \rightarrow R(x))$ and $\forall x(T(x) \rightarrow W(x))$.

D is inconsistent wrt IC, because tuples $S(a), T(a)$ belong to D, but $Q(a), W(a)$ are not in D. There are four repairs:

1. $\{T(a),\, W(a),\, S(a),\, S(b),\, Q(b),\, R(b),\, Q(a), R(a)\}$
2. $\{S(a),\, S(b),\, Q(b),\, R(b),\, Q(a),\, R(a)\}$
3. $\{T(a),\, W(a),\, S(b),\, Q(b),\, R(b)\}$
4. $\{S(b),\, Q(b), R(b)\}$

For query $\mathcal{Q}$: $Ans(x) \leftarrow S(x)$ the consistent answer is (b). □

Figure 9.1 shows the architecture of *ConsEx*.

The database parameters are received by the *Database Connection* module, which gets connected with the database instance.

The *Query Processing* module is the central module in *ConsEx*. It coordinates all the tasks that are needed to compute the consistent answers to queries. This module receives the F0 queries, the ICs, and the options of the methodology selected by the user to evaluate the queries. In order to compute consistent answers, this module interacts with other modules which perform specific tasks.

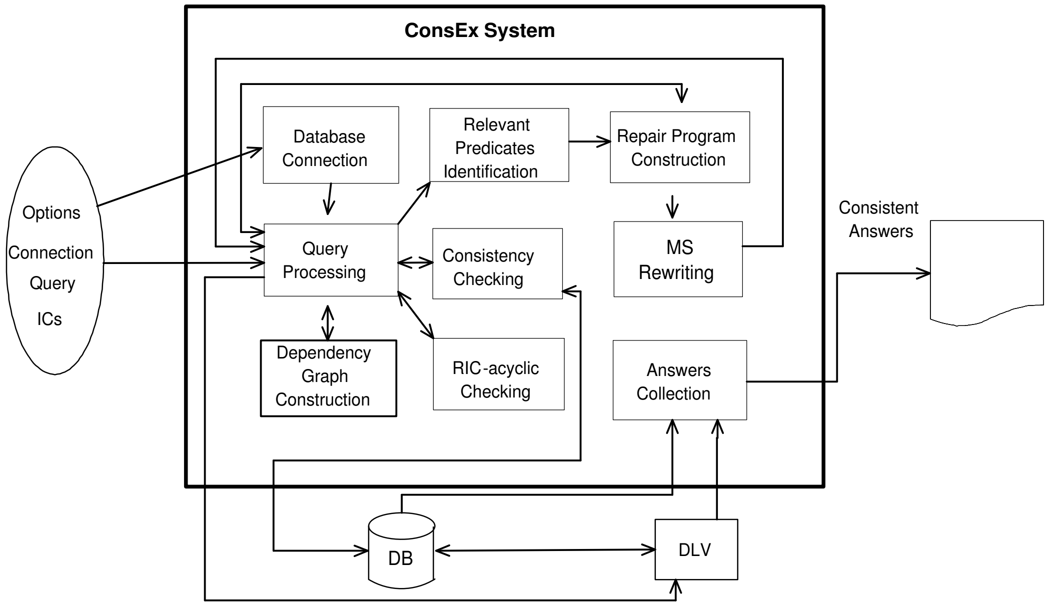


Figure 9.1: ConsEx Architecture

The sequence of tasks performed by the *Query Processing* module are:

(1) Sintaxis checking. It checks that queries are syntactically correct. *ConsEx* supports any F0 query expressed as a DLV sentence [61].[1] Also, *ConsEx* supports SQL queries that are conjunction of atoms, but not disjunctive SQL queries.

Query $Ans(x) \leftarrow S(x)$ in Example 9.1 is correct, i.e. it is a well-written sentence. Also, it queries predicates S which is a valid database relation.

(2) Translation of queries. It translates the FO queries into logic program (cf. Chapter 2). Likewise, if the query is a SQL query, it is first translated into an equivalent Datalog program [64], and then into a logic program.

For query $Ans(x) \leftarrow S(x)$, the query program $\Pi(\mathcal{Q})$ contains rule:

$$Ans(x) \leftarrow \underline{S}(x, \mathbf{t}^{\star\star}).$$

[1]In DLV, a rule of the form $P(x) \leftarrow Q(x)$ will be written as $P(x) :- Q(x)$, i.e. the implication symbol "$\leftarrow$" is replaced by symbol "$:-$". In this chapter, and for presentation purposes, we will use symbol "$\leftarrow$" instead of "$:-$".

(3) Generation of the dependency graph. It calls to the *Dependency Graph Construction* module with the set of ICs, which generates the dependency graph (cf. Definition 2.1). This graph is used later to identify the relevant predicates to compute queries, and also to generate the program constraints for repair programs.

Figure 9.2 shows the dependency graph $\mathcal{G}(IC)$ for the ICs in Example 9.1.

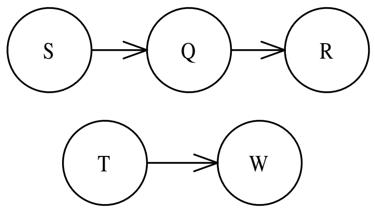


Figure 9.2: Dependency Graph $\mathcal{G}(IC)$ for Example 9.1

(4) Identification of relevant predicates. It calls to the *Relevant Predicates Identification* module with the query and the dependency graph. This module identifies the relevant predicates to compute the query, which is achieved by examining the interaction between predicates in the dependency graph, and the query predicates, i.e. those database predicates in the query (cf. Section 5.4).

For query $Ans(x) \leftarrow S(x)$ and dependency graph $\mathcal{G}(IC)$, the relevant predicates are S, Q, R.

The identification of the relevant predicates is important to determine if the query can be answered directly on the database. This is done next in the *Query Processing* module.

(5) Direct computation of queries. The *Query Processing* module also checks if the query can be computed directly in the database instance, without generating any repair program. In order to do that, it calls to the *Consistency Checking* module with the query, and the dependency graph restricted to the predicates (nodes) that are relevant to compute the query.

Figure 9.3 shows the dependency graph restricted to the relevant predicates to compute query $Ans(x) \leftarrow S(x)$.

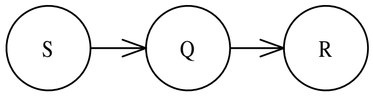


Figure 9.3: Dependency Graph $\mathcal{G}(IC)$ Restricted to the Relevant Predicates

The *Consistency Checking* module detects if the database instance violates the ICs involving predicates that are relevant to compute the query. In order to do that, it generated an ah-doc SQL query, for each relevant IC, that checks the satisfiability of the IC in the database. If the database is consistent wrt the relevant ICs, then the query can be answered directly by the DBMS; otherwise the repair program has to be generated.

The relevant ICs for query $Ans(x) \leftarrow S(x)$ are:

- $\forall x(S(x) \rightarrow Q(x))$.
- $\forall x(Q(x) \rightarrow R(x))$.

The *Consistency Checking* module generates the following SQL queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ to check, respectively, consistency of the relevant ICs:

```
SELECT S.ID
FROM S
WHERE S.ID NOT IN (SELECT S.ID
                   FROM S, Q
                   WHERE S.ID = Q.ID)
```

```
SELECT Q.ID
FROM Q
WHERE Q.ID NOT IN (SELECT Q.ID
                   FROM Q, R
                   WHERE Q.ID = R.ID)
```

Since the evaluation of $\mathcal{Q}_1$ in the database instance D returns tuple (a), D does not satisfy the IC $\forall x(S(x) \rightarrow Q(x))$. As a consequence, the query $Ans(x) \leftarrow S(x)$ cannot be answered directly in the database, and the repair program has to be generated.

Notice that in absence of atom $S(a)$, the database instance is consistent wrt the relevant ICs, and query $Ans(x) \leftarrow S(x)$ can be answered directly in the database. In this case, the *Consistency Checking* module will generate the SQL query:

"`SELECT ID FROM S`",

which is evaluated directly in the database instance and retrieves the consistent answers to $\mathcal{Q}$.

(6) Checking RIC-acyclic property. If the database is inconsistent wrt the relevant ICs to compute the query, then the repair program has to be generated. Therefore, it is necessary to check if the set of ICs is RIC-acyclic. In order to do that, the *Query Processing* module calls to the *RIC-acyclic Checking* module, which checks if the ICs are RIC-acyclic.

The *RIC-acyclic Checking* module receives the dependency graph $\mathcal{G}(IC)$, generates the *contracted dependency graph* (cf. Definition 2.2), and checks if there are cycles in it. If the contracted dependency graph has cycles, then the set of ICs is not RIC-acyclic.

Figure 9.4 shows the contracted dependency graph for $\mathcal{G}(IC)$.

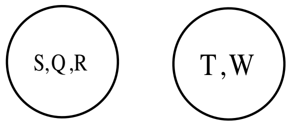


Figure 9.4: Contracted Dependency Graph $\mathcal{G}^C(IC)$ for Example 9.1

Since there are no loops in $\mathcal{G}^C(IC)$, the set IC in Example 9.1 is RIC-acyclic.

This property has to be checked since the specification of repair programs implemented in the system works for RIC-acyclic sets of UICs, RICs, and NNCs. If the set of ICs is non-RIC-acyclic, the generation of repair programs is avoided, and a warning message is sent to the user.

(7) Evaluation of queries by using MS. If the user selects MS to compute the query, the *Query Processing* module first calls to the *Repair Program Construction* module, with the set of ICs, which generates the repair program (cf. Definition 4.2).

Repair programs are constructed "on the fly", that is, all the annotations that appear in the programs are generated by the system, and the database is not affected. Also, database facts are not imported into the *ConsEx* system. Instead, suitable sentences to import data are added to a repair program. As an illustration, for the database instance D in Example 9.1, the repair program will contain sentences of the form:

$$\#\mathit{import}(\mathit{dbName}, \text{“}\mathit{dbUser}\text{”}, \text{“}\mathit{dbPass}\text{”}, \text{“}\mathit{SELECT\ DISTINCT} * \mathit{FROM\ P}\text{”}, P) \tag{9.1}$$

where *dbName* is the database name, *dbUser* is the database user, and *dbPass* is the user password. The SQL statement collects the tuples stored in relation P. Note that by using "*DISTINCT*" in the query, we do not import multiples copies of a same tuple stored in relation P. The last parameter is the name of the predicate that will contain, in the repair program, the tuples retrieved from relation P.

As a result, when the program is evaluated in the DLV system, the database facts are imported directly to the reasoning system.

Program 9.1 is the repair program generated by the *Repair Program Construction* module, for the database instance and ICs in Example 9.1. We assume the following parameters:

- Database name: *test*.
- Database user and password: *user01*.

Program 9.1

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ \ DISTINCT * FROM\, S\text{”}, S)$

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ \ DISTINCT * FROM\, Q\text{”}, Q)$

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ \ DISTINCT * FROM\, R\text{”}, R)$

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ \ DISTINCT * FROM\, T\text{”}, T)$

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ \ DISTINCT * FROM\, W\text{”}, W)$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow S\_(x, \mathbf{t}^\star), Q\_(x, \mathbf{f_a}),\ x \neq null.$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow S\_(x, \mathbf{t}^\star),\ \ not\ \ Q(x),\ x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow Q\_(x, \mathbf{t}^\star), R\_(x, \mathbf{f_a}),\ x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow Q\_(x, \mathbf{t}^\star),\ \ not\ \ R(x),\ x \neq null.$

$T\_(x, \mathbf{f_a}) \vee W\_(x, \mathbf{t_a}) \leftarrow T\_(x, \mathbf{t}^\star), W\_(x, \mathbf{f_a}),\ x \neq null.$

$T\_(x, \mathbf{f_a}) \vee W\_(x, \mathbf{t_a}) \leftarrow T\_(x, \mathbf{t}^\star),\ \ not\ \ W(x),\ x \neq null.$

$$\left.\begin{array}{l} S\_(x, \mathbf{t}^\star) \leftarrow S\_(x, \mathbf{t_a}). \\ S\_(x, \mathbf{t}^\star) \leftarrow S(x). \\ S\_(x, \mathbf{t}^{\star\star}) \leftarrow S\_(x, \mathbf{t}^\star),\ \ not\ \ S\_(x, \mathbf{f_a}). \end{array}\right\} \textit{(Similarly for } Q, R, T \textit{ and } W\textit{)}$$

$\leftarrow Q\_(x, \mathbf{t_a}), Q\_(x, \mathbf{f_a}).$ □

Repair programs are stored for later computations. This is possible since repair programs depend on the ICs and database relations, but not on queries. Therefore, repair programs do not need to be generated every time a query is processed. In fact, they are re-generated only when the ICs or the data schema are modified.

After the repair program is generated, the *Query Processing* module calls to the *MS Rewriting* module with the query and repair programs. The latter module generates the MS rewritten program (cf. Chapter 5).

Program 9.2 is the magic program generated by the *MS Rewriting* module, for the repair program in 9.1 and the query program $Ans(x) \leftarrow S\_(x, \mathbf{t}^{\star\star})$.

Program 9.2

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ DISTINCT * FROM\, S\text{”}, S)$

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ DISTINCT * FROM\, Q\text{”}, Q)$

$\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ DISTINCT * FROM\, R\text{”}, R)$

$Ans(x) \leftarrow magic\_Ans^f, S\_(x, \mathbf{t}^{\star\star}).$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}), magic\_Q\_^{fb}(\mathbf{t_a}), S\_(x, \mathbf{t}^\star), Q\_(x, \mathbf{f_a}), x \neq null.$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}), magic\_Q\_^{fb}(\mathbf{t_a}), S\_(x, \mathbf{t}^\star),\ not\ Q(x), x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{f_a}), magic\_R\_^{fb}(\mathbf{t_a}), Q\_(x, \mathbf{t}^\star), R\_(x, \mathbf{f_a}), x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{f_a}), magic\_R\_^{fb}(\mathbf{t_a}), Q\_(x, \mathbf{t}^\star),\ not\ R(x), x \neq null.$

$S\_(x, \mathbf{t}^\star) \leftarrow magic\_S\_^{fb}(\mathbf{t}^\star), S\_(x, \mathbf{t_a}). \quad S\_(x, \mathbf{t}^\star) \leftarrow magic\_S\_^{fb}(\mathbf{t}^\star), S(x).$

$Q\_(x, \mathbf{t}^\star) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^\star), Q\_(x, \mathbf{t_a}). \quad Q\_(x, \mathbf{t}^\star) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^\star), Q(x).$

$R\_(x, \mathbf{t}^\star) \leftarrow magic\_R\_^{fb}(\mathbf{t}^\star), R\_(x, \mathbf{t_a}). \quad R\_(x, \mathbf{t}^\star) \leftarrow magic\_R\_^{fb}(\mathbf{t}^\star), R(x).$

$S\_(x, \mathbf{t}^{\star\star}) \leftarrow magic\_S\_^{fb}(\mathbf{t}^{\star\star}), S\_(x, \mathbf{t}^\star),\ not\ S\_(x, \mathbf{f_a}).$

$Q\_(x, \mathbf{t}^{\star\star}) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^{\star\star}), Q\_(x, \mathbf{t}^\star),\ not\ Q\_(x, \mathbf{f_a}).$

$R\_(x, \mathbf{t}^{\star\star}) \leftarrow magic\_R\_^{fb}(\mathbf{t}^{\star\star}), R\_(x, \mathbf{t}^\star),\ not\ R\_(x, \mathbf{f_a}).$

$magic\_S\_^{fb}(\mathbf{t}^{\star\star}) \leftarrow magic\_Ans^f.$

$magic\_Q\_^{fb}(\mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}).$

$magic\_S\_^{fb}(\mathbf{t}^\star) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}).$

$magic\_Q\_^{fb}(\mathbf{f_a}) \leftarrow magic\_S\_^{fb}(\mathbf{f_a}).$

$magic\_S\_^{fb}(\mathbf{f_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{t_a}).$

$magic\_S\_^{fb}(\mathbf{t}^\star) \leftarrow magic\_Q\_^{fb}(\mathbf{t_a}).$

$magic\_Q\_^{fb}(\mathbf{f_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{t_a}).$

$magic\_S\_^{fb}(\mathbf{t_a}) \leftarrow magic\_S\_^{fb}(\mathbf{t}^\star).$

$magic\_Q\_^{fb}(\mathbf{t_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^\star).$

$magic\_S\_^{fb}(\mathbf{t}^\star) \leftarrow magic\_S\_^{fb}(\mathbf{t}^{\star\star}).$

$magic\_S\_^{fb}(\mathbf{f_a}) \leftarrow magic\_S\_^{fb}(\mathbf{t}^{\star\star}).$

$magic\_Q\_^{fb}(\mathbf{t}^\star) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^{\star\star}).$

$magic\_Q\_^{fb}(\mathbf{f_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{t}^{\star\star}).$

$magic\_R\_^{fb}(\mathbf{t_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{f_a}).$

$magic\_Q\_^{fb}(\mathbf{t}^\star) \leftarrow magic\_Q\_^{fb}(\mathbf{f_a}).$

$magic\_R\_^{fb}(\mathbf{f_a}) \leftarrow magic\_Q\_^{fb}(\mathbf{f_a}).$

$magic\_Q\_^{fb}(\mathbf{f_a}) \leftarrow magic\_R\_^{fb}(\mathbf{t_a}).$

$magic\_Q\_^{fb}(\mathbf{t}^\star) \leftarrow magic\_R\_^{fb}(\mathbf{t_a}).$

$magic\_R\_^{fb}(\mathbf{f_a}) \leftarrow magic\_R\_^{fb}(\mathbf{t_a}).$

$magic\_R\_^{fb}(\mathbf{t_a}) \leftarrow magic\_R\_^{fb}(\mathbf{t}^\star).$

$magic\_R\_^{fb}(\mathbf{t}^\star) \leftarrow magic\_R\_^{fb}(\mathbf{t}^{\star\star}).$

$magic\_R\_^{fb}(\mathbf{f_a}) \leftarrow magic\_R\_^{fb}(\mathbf{t}^{\star\star}).$

$magic\_Ans^f.$

$\leftarrow Q\_(x, \mathbf{t_a}), Q\_(x, \mathbf{f_a}).$ □

Notice that in Program 9.2 only the relevant portion of the database is imported to DLV. This is, only tuples for relations S, Q and R, which are the relevant predicates to compute the query, are imported to DLV. Moreover, since there are no constants in the query, all the magic atoms appear with annotation "*fb*" in the MS program. For instance, in the modified rule:

$$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow \mathit{magic\_S\_}^{fb}(\mathbf{f_a}), \mathit{magic\_Q\_}^{fb}(\mathbf{t_a}), S\_(x, \mathbf{t}^{\star}), Q\_(x, \mathbf{f_a}), x \neq \mathit{null},$$

the magic atom $\mathit{magic\_S\_}^{fb}(\mathbf{f_a})$ establishes that the first attribute of "$S\_$" is free, and the second, which corresponds to the constant $\mathbf{f_a}$ is bound. Therefore, the import sentence for predicate S imports all the tuples from relation S. This also happens for the relevant predicates Q and R.

For a ground (or partially-ground) query like $Ans \leftarrow S(b)$, the MS program would import a more reduced amount of tuples. This is because the constants in the query would be used to restrict the tuples involve in the computation of the query (cf. Chapter 5). As an illustration, for query $Ans \leftarrow S(b)$, again the relevant predicates to compute the query are S, Q and R, which share the attribute *ID*. Since the query asks for a specific value in the attribute *ID*, the corresponding MS program will push-down the constant "*b*" into the rules containing predicates S, Q, and R. Then, the following import sentences will be generated in the MS program (assuming the same database parameters):

- $\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ \ DISTINCT * FROM\ \ S$
  $WHERE\ \ ID = \text{‘}b\text{‘”}, S)$

- $\#import(test, \text{“}user01\text{”}, \text{“}user01\text{”}, \text{“}SELECT\ \ DISTINCT * FROM\ \ Q$
  $WHERE\ \ ID = \text{‘}b\text{‘”}, Q)$

- $\#import(test, \text{``}user01\text{''}, \text{``}user01\text{''}, \text{``}SELECT\ DISTINCT * FROM\ R$
  $WHERE\ ID = \text{`}b\text{`''}, R)$

In Section 9.4 we present more examples of ground (and partially-ground) conjunctive queries, and also we show that the selection of tuples performed by MS has a direct relation with the good performance of MS to evaluate queries.

After creating the MS rewritten program, the *Query Processing* module sends it to DLV where it is evaluated. It is important to remark that DLV will import the database facts directly from the database instance. In this way, facts are never bringing to the main memory, and the flow of data between the *ConsEx* system and the database is reduced.

DLV returns the query answers to the *Answer Collection* module. This module, collect answers, performs any formatting needed, and constructs a final answer to be returned to the user. For the Program 9.2, DLV returns (b) as the consistent answer to the query. This answer is sent to the *Answer Collection* module, which constructs the final answer:

```
 ID
---
 b
```

which is displayed in the user interface.

(8) Evaluation of queries by using the RP methodology. If the user selects the RP methodology instead of MS, the *Query Processing* module calls to the *Repair Program Construction* module with the relevant predicates to compute the query. In this manner, the repair program is generated for the relevant predicates, only (cf. Section 5.4).

Program 9.3 is the repair program for predicates S, Q, R, which are the relevant predicates to compute query $Ans(x) \leftarrow S\_(x, \mathbf{t}^{\star\star})$.

Program 9.3

$\#import(test, \text{"}user01\text{"}, \text{"}user01\text{"}, \text{"}SELECT\ \ DISTINCT * FROM\, S\text{"}, S)$

$\#import(test, \text{"}user01\text{"}, \text{"}user01\text{"}, \text{"}SELECT\ \ DISTINCT * FROM\, Q\text{"}, Q)$

$\#import(test, \text{"}user01\text{"}, \text{"}user01\text{"}, \text{"}SELECT\ \ DISTINCT * FROM\, R\text{"}, R)$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow S\_(x, \mathbf{t}^\star), Q\_(x, \mathbf{f_a}),\ x \neq null.$

$S\_(x, \mathbf{f_a}) \vee Q\_(x, \mathbf{t_a}) \leftarrow S\_(x, \mathbf{t}^\star),\ \ not\ \ Q(x),\ x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow Q\_(x, \mathbf{t}^\star), R\_(x, \mathbf{f_a}),\ x \neq null.$

$Q\_(x, \mathbf{f_a}) \vee R\_(x, \mathbf{t_a}) \leftarrow Q\_(x, \mathbf{t}^\star),\ \ not\ \ R(x),\ x \neq null.$

$$\left.\begin{array}{l} S\_(x, \mathbf{t}^\star) \leftarrow S\_(x, \mathbf{t_a}). \\ S\_(x, \mathbf{t}^\star) \leftarrow S(x). \\ S\_(x, \mathbf{t}^{\star\star}) \leftarrow S\_(x, \mathbf{t}^\star),\ \ not\ \ S\_(x, \mathbf{f_a}). \end{array}\right\} \textit{(Similarly for } Q \textit{ and } R\textit{)}$$

$\leftarrow Q\_(x, \mathbf{t_a}), Q\_(x, \mathbf{f_a}).$ □

The *Query Processing* module sends the new repair program together with the query program to DLV, which returns the consistent answer (b) to the *Answer Collection* module. The latter constructs the final answer, which is displayed in the user interface.

In the next section we present the user interface of the system.

9.3 Graphical User Interface

Figure 9.5 is the connection screen in *ConsEx*. It receives the database parameters, i.e. the database name, ip (for remote connections), user, and password. In this manner, *ConsEx* gets connected with the database instance.

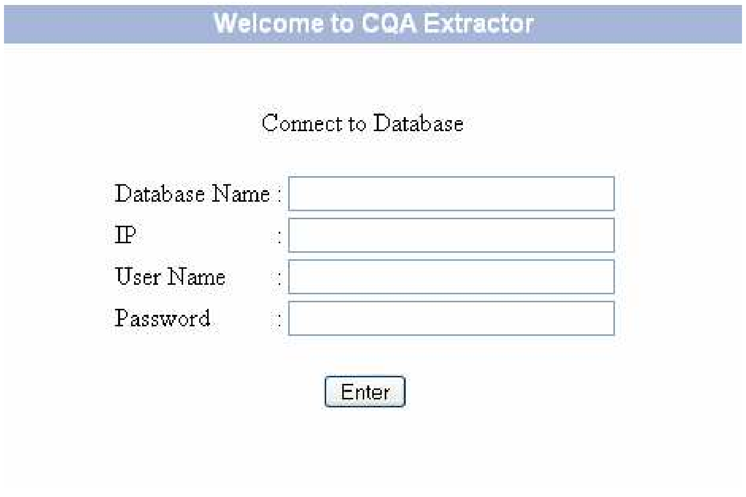


Figure 9.5: Database Connection in *ConsEx*

When the system is connected to the database, the main menu in Figure 9.6 is displayed. The main menu consists of five options.

The first option is “DB Edition”, which allows the edition of the data stored in the database. Here, the user can add new relations, and tuples; eliminate relations and tuples; and update relations. Figure 9.7 shows information about the database instance in Example 9.1.

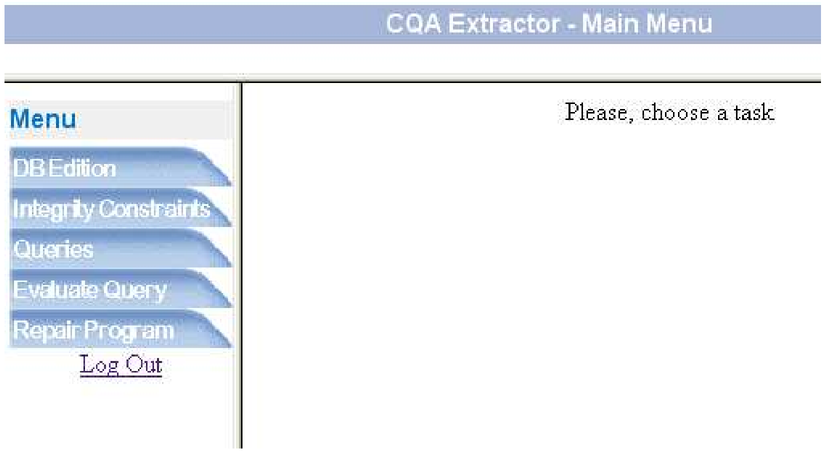
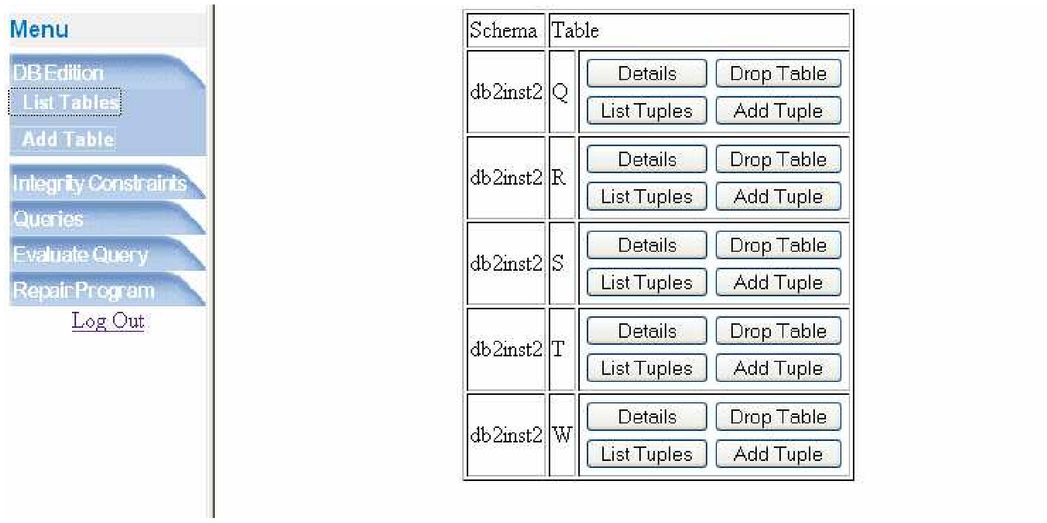


Figure 9.6: Main Menu in *ConsEx*

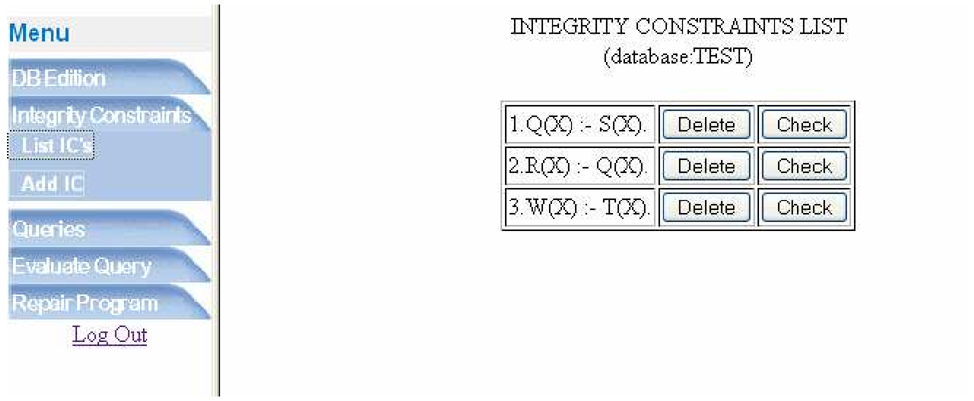
The second option in *ConsEx* is “Integrity Constraints”. Here the user can list, add and eliminate integrity constraints (*ConsEx* supports UICs, RICs, and NNCs). Figure 9.8 shows the ICs in Example 9.1.[2]



Schema	Table			
db2inst2	Q	Details	Drop Table	
		List Tuples	Add Tuple	
db2inst2	R	Details	Drop Table	
		List Tuples	Add Tuple	
db2inst2	S	Details	Drop Table	
		List Tuples	Add Tuple	
db2inst2	T	Details	Drop Table	
		List Tuples	Add Tuple	
db2inst2	W	Details	Drop Table	
		List Tuples	Add Tuple	

Figure 9.7: Database Edition in *ConsEx*

We can see that the system allows to check if a specific IC is satisfied by the database instance. In Figure 9.8 the button “Check” besides every IC performs the checking of the corresponding IC.



INTEGRITY CONSTRAINTS LIST
(database:TEST)

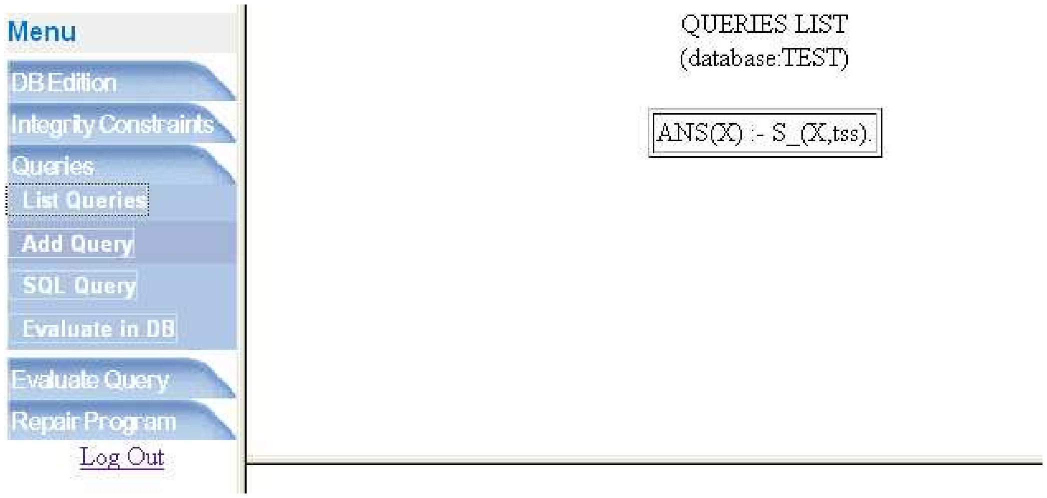
1.Q(X) :- S(X).	Delete	Check
2.R(X) :- Q(X).	Delete	Check
3.W(X) :- T(X).	Delete	Check

Figure 9.8: Integrity Constraints in Example 9.1

The third option in *ConsEx* is “Queries”. Here, the user can list and add new

[2]In *ConsEx*, ICs and queries (but not SQL queries) are written as DLV sentences, then it uses symbol “ $:-$ ” instead of symbol “ $\leftarrow$ ”.

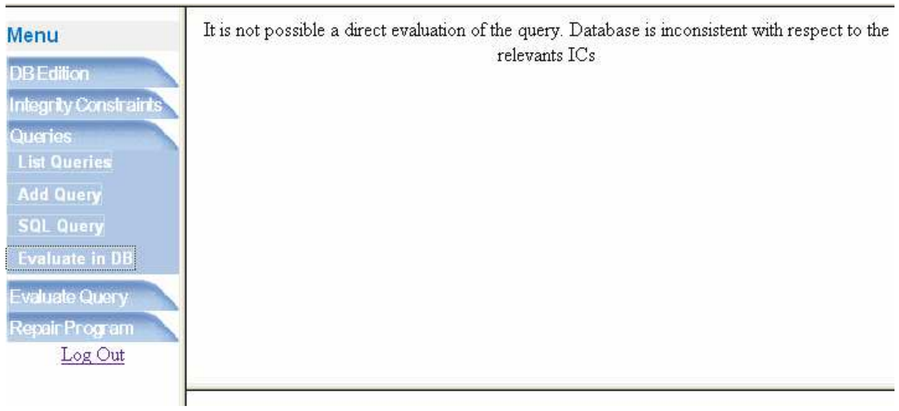
queries. Also, there is an option to evaluate queries directly on the database, which will be allow only if the database satisfy the relevant ICs to compute the query. Figure 9.9 shows the query program for query $Ans(x) \leftarrow S(x)$ in Example 9.1.



QUERIES LIST
(database:TEST)

ANS(X) :- S_(X,tss).

Figure 9.9: Query in Example 9.1

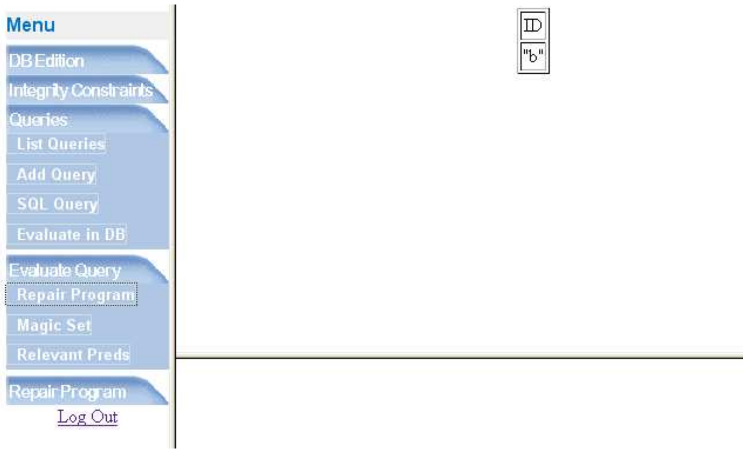


It is not possible a direct evaluation of the query. Database is inconsistent with respect to the relevants ICs

Figure 9.10: Warning Message for the Query in Example 9.1

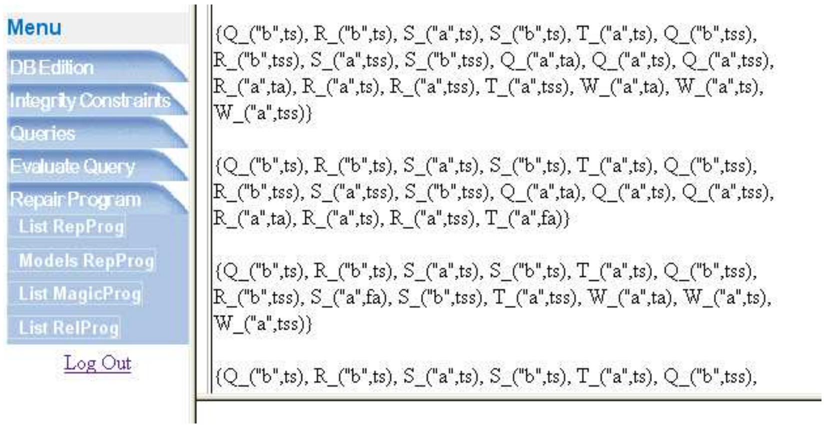
If the user wants to evaluate the query $Ans(x) \leftarrow S(x)$ directly in the database, the system will send the warning message in Figure 9.10. This is because, since the database is inconsistent wrt the relevant ICs (cf. Example 9.1), the query cannot be answered directly in the database, and the repair program has to be generated.

The four option in the system is “Evaluate Query”. Here, the system allows to evaluate queries using the MS or the RP methodologies. Also, and just for experimental purposes, the system allows the straightforward evaluation of the query and repair programs. As an illustration, Figure 9.11 shows the consistent answer for query $Ans(x) \leftarrow S(x)$ in Example 9.1 obtained with the MS methodology.



```
ID
"b"
```

Figure 9.11: Consistent Answer for the Query in Example 9.1



```
{Q_("b",ts), R_("b",ts), S_("a",ts), S_("b",ts), T_("a",ts), Q_("b",tss),
R_("b",tss), S_("a",tss), S_("b",tss), Q_("a",ta), Q_("a",ts), Q_("a",tss),
R_("a",ta), R_("a",ts), R_("a",tss), T_("a",tss), W_("a",ta), W_("a",ts),
W_("a",tss)}

{Q_("b",ts), R_("b",ts), S_("a",ts), S_("b",ts), T_("a",ts), Q_("b",tss),
R_("b",tss), S_("a",tss), S_("b",tss), Q_("a",ta), Q_("a",ts), Q_("a",tss),
R_("a",ta), R_("a",ts), R_("a",tss), T_("a",fa)}

{Q_("b",ts), R_("b",ts), S_("a",ts), S_("b",ts), T_("a",ts), Q_("b",tss),
R_("b",tss), S_("a",fa), S_("b",tss), T_("a",tss), W_("a",ta), W_("a",ts),
W_("a",tss)}

{Q_("b",ts), R_("b",ts), S_("a",ts), S_("b",ts), T_("a",ts), Q_("b",tss),
```

Figure 9.12: Stable Models for the Repair Program in Program 9.1

The last option in *ConsEx* is “Repair Program” which lists the repair program, the MS rewritten program, and the repair program generated by RP. Also, it permits to display the stable models of the repair program. Figure 9.12 displays the stable models of the repair program in Program 9.1.

9.4 Experimental Evaluation

In this section we analyze the results of the experiments we performed to quantify the gain, in terms of the execution time of queries, of the methods implemented in *ConsEx*.

9.4.1 Experimental Setup

The experiments were performed on a Intel Pentium 4 PC, processor of 3.00 Ghz, 512 MB of RAM, and with Linux distribution UBUNTU 6.0. The database instances were stored on DB2 Universal Database Server Edition, version 8.2 for Linux platforms. All the programs were run in the DLV prototype for Linux distribution released on Jan 12, 2006.

For the experiments we use database instances D_1 and D_2 for an airport system, with database schema composed by the following relations:

- *Passenger (PID, PNAME,PHONE)*. It stores the ID, name and phone number of passengers.

- *Luggage(TAG, WEIGHT)*. It stores the tag number, and weight of luggages.

- *Planetype(TID, TNAME, MAXPASS, MAXLUGGWEIGHT)*. It stores the ID, name, maximum of passengers and weight of plane types.

- *Plane(PID, TYPE)*. It stores the ID, and plane type of planes.

- *Inspection(IID, DATE, TIME, PID, STATUS)*. It stores the ID, date, time, plane ID, and status of inspections performed to planes.

- *Flight(FNUM, DEPART, ARR, DATE, ORIGIN, DESTINATION, GATE, ASEATS, PID)*. It stores the number, departure and arrival times, date, cities of origin and destination, gate, available seats, and plane ID of flights.
- *Flying(PID, TAG, FNUM)*. It stores passenger IDs, tags and flight numbers.
- *Booking(BID, PID, FNUM, SNUMBER)*. It stores booking IDs, passengers IDs, flights and seats numbers.

The set IC of ICs is:

1. $\forall xyzsw(\mathit{Passenger}(x,y,z) \wedge \mathit{Passenger}(x,s,w) \rightarrow y=s)$
2. $\forall xyzsw(\mathit{Passenger}(x,y,z) \wedge \mathit{Passenger}(x,s,w) \rightarrow z=w)$
3. $\forall xyz(\mathit{Luggage}(x,y) \wedge \mathit{Luggage}(x,z) \rightarrow y=z)$
4. $\forall xyzwsmu(\mathit{Planetype}(x,y,z,w) \wedge \mathit{Planetype}(x,s,m,u) \rightarrow y=s)$
5. $\forall xyzwsmu(\mathit{Planetype}(x,y,z,w) \wedge \mathit{Planetype}(x,s,m,u) \rightarrow z=m)$
6. $\forall xyzwsmu(\mathit{Planetype}(x,y,z,w) \wedge \mathit{Planetype}(x,s,m,u) \rightarrow w=u)$
7. $\forall xyz(\mathit{Plane}(x,y) \wedge \mathit{Plane}(x,z) \rightarrow y=z)$
8. $\forall xyzws(\mathit{Booking}(x,y,z,w) \wedge \mathit{Booking}(x,y,z,s) \rightarrow w=s)$
9. $\forall xyzwsmurt(\mathit{Inspection}(x,y,z,w,s) \wedge \mathit{Inspection}(x,m,u,r,t) \rightarrow y=m)$
10. $\forall x0x1x2x3x4x5x6x7x8x9x10x11x12x13x14x15x16x17\ (\mathit{Flight}(x0,x1,x2,x3,x4,x5,x6,x7,x8) \wedge \mathit{Flight}(x0,x10,x11,x12,x13,x14,x15,x16,x17) \rightarrow x1 = x10)$
11. $\forall zx(\mathit{Plane}(z,x) \rightarrow \exists yuw(\mathit{Planetype}(x,y,u,w))$

12. $\forall xyzws(\mathit{Inspection}(x,y,z,w,s) \rightarrow \exists u(\mathit{Plane}(w,u))$

13. $\forall x0x1x2x3x4x5x6x7x8\ (\mathit{Flight}(x0,x1,x2,\ x3,x4,x5,x6,x7,x8) \rightarrow$
$\exists z\ \mathit{Plane}(x8,z))$

ICs $1-8$ specify the primary keys for relations *Passenger, Luggage, Planetype, Plane,* and *Booking.* ICs $9-10$ are FDs over relations *Inspection* and *Flight.* ICs $11-13$ are RICs.

Figure 9.13 is the dependency graph $\mathcal{G}(IC)$ for the set of ICs.

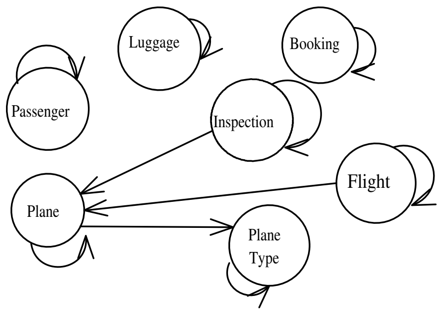


Figure 9.13: Dependency Graph $\mathcal{G}(IC)$ for the ICs in the Experimental Database

Assuming the database name *airport*, and database user and password *db2inst2*, the repair program for the database instance D_1 (also D_2) contains the rules in Program 9.4.

Program 9.4

$$\left.\begin{array}{l}
\#\mathit{import}(\mathit{airport}, \text{“}\mathit{db2inst2}\text{”}, \text{“}\mathit{db2inst2}\text{”},\\
\qquad\qquad \text{“}\mathit{SELECT\ \ DISTINCT}*\\
\qquad\qquad \mathit{FROM\ \ Passenger}\text{”}, \mathit{Passenger}).\\
\mathit{Passenger\_}(x,y,z,\mathbf{t}^\star) \leftarrow \mathit{Passenger\_}(x,y,z,\mathbf{t_a}).\\
\mathit{Passenger\_}(x,y,z,\mathbf{t}^\star) \leftarrow \mathit{Passenger}(x,y,z).\\
\mathit{Passenger\_}(x,y,z,\mathbf{t}^{\star\star}) \leftarrow \mathit{Passenger\_}(x,y,z,\mathbf{t}^\star),\\
\qquad\qquad\qquad \mathit{not\ \ Passenger\_}(x,y,z,\mathbf{f_a}).
\end{array}\right\}\ \mathit{(For\ each\ relation\ in\ D_1)}$$

$$Passenger\_(x, y, z, \mathbf{f_a}) \vee Passenger\_(x, s, w, \mathbf{f_a}) \leftarrow Passenger\_(x, y, z, \mathbf{t^\star}),$$
$$Passenger\_(x, s, w, \mathbf{t^\star}), y \neq s, x \neq null, s \neq null, y \neq null.$$

$$Passenger\_(x, y, z, \mathbf{f_a}) \vee Passenger\_(x, s, w, \mathbf{f_a}) \leftarrow Passenger\_(x, y, z, \mathbf{t^\star}),$$
$$Passenger\_(x, s, w, \mathbf{t^\star}), z \neq w, x \neq null, w \neq null, z \neq null.$$

$$Luggage\_(x, y, \mathbf{f_a}) \vee Luggage\_(x, z, \mathbf{f_a}) \leftarrow Luggage\_(x, y, \mathbf{t^\star}),$$
$$Luggage\_(x, z, \mathbf{t^\star}), y \neq z, x \neq null, z \neq null, y \neq null.$$

$$Planetype\_(x, y, z, w, \mathbf{f_a}) \vee Planetype\_(x, s, m, u, \mathbf{f_a}) \leftarrow Planetype\_(x, y, z, w, \mathbf{t^\star}),$$
$$Planetype\_(x, s, m, u, \mathbf{t^\star}),\ y \neq s, x \neq null, s \neq null, y \neq null.$$

$$Planetype\_(x, y, z, w, \mathbf{f_a}) \vee Planetype\_(x, s, m, u, \mathbf{f_a}) \leftarrow Planetype\_(x, y, z, w, \mathbf{t^\star}),$$
$$Planetype\_(x, s, m, u, \mathbf{t^\star}),\ z \neq m, x \neq null, m \neq null, z \neq null.$$

$$Planetype\_(x, y, z, w, \mathbf{f_a}) \vee Planetype\_(x, s, m, u, \mathbf{f_a}) \leftarrow Planetype\_(x, y, z, w, \mathbf{t^\star}),$$
$$Planetype\_(x, s, m, u, \mathbf{t^\star}),\ w \neq u, x \neq null, u \neq null, w \neq null.$$

$$Plane\_(x, y, \mathbf{f_a}) \vee Plane\_(x, z, \mathbf{f_a}) \leftarrow Plane\_(x, y, \mathbf{t^\star}), Plane\_(x, z, \mathbf{t^\star}),$$
$$y \neq z, x \neq null, z \neq null, y \neq null.$$

$$Booking\_(x, y, z, w, \mathbf{f_a}) \vee Booking\_(x, y, z, s, \mathbf{f_a}) \leftarrow Booking\_(x, y, z, w, \mathbf{t^\star}),$$
$$Booking\_(x, y, z, s, \mathbf{t^\star}), w \neq s, x \neq null, y \neq null, z \neq null, s \neq null,\ w \neq null.$$

$$Inspection\_(x, y, z, w, s, \mathbf{f_a}) \vee Inspection\_(x, m, u, r, t, \mathbf{f_a}) \leftarrow Inspection\_(x, y, z, w, s, \mathbf{t^\star}),$$
$$Inspection\_(x, m, u, r, t, \mathbf{t^\star}), y \neq m,\ x \neq null, m \neq null, y \neq null.$$

$$Flight\_(x0, x1, x2, x3, x4, x5, x6, x7, x8, \mathbf{f_a}) \vee$$
$$Flight\_(x0, x10, x11, x12, x13, x14, x15, x16, x17, \mathbf{f_a}) \leftarrow$$
$$Flight\_(x0, x1, x2, x3, x4, x5, x6, x7, x8, \mathbf{t^\star}), Flight\_(x0, x10, x11, x12, x13, x14, x15, x16,$$
$$x17, \mathbf{t^\star}), x1 \neq x10, x0 \neq null, x10 \neq null.$$

$$Plane\_(z, x, \mathbf{f_a}) \vee Planetype\_(x, null, null, null, \mathbf{t_a}) \leftarrow Plane\_(z, x, \mathbf{t^\star}),\ \ not\ \ aux1\_(x),$$
$$x \neq null.$$

$$aux1\_(x) \leftarrow Planetype\_(x, y, U, w, \mathbf{t^\star}),\ not\ Planetype\_(x, y, u, w, \mathbf{f_a}), x \neq null, y \neq null.$$

$$aux1\_(x) \leftarrow Planetype\_(x, y, u, w, \mathbf{t^\star}),\ not\ Planetype\_(x, y, u, w, \mathbf{f_a}), x \neq null, u \neq null.$$

$aux1\_(x) \leftarrow Planetype\_(x, y, u, w, \mathbf{t^\star}),\ not\ Planetype\_(x, y, u, w, \mathbf{f_a}), x \neq null, w \neq null.$

$Inspection\_(x, y, z, w, s, \mathbf{f_a}) \vee Plane\_(w, null, \mathbf{t_a}) \leftarrow Inspection\_(x, y, z, w, s, \mathbf{t^\star}),$

$not\ aux2\_(w), w \neq null.$

$aux2\_(w) \leftarrow Plane\_(w, u, \mathbf{t^\star}),\ not\ Plane\_(w, u, \mathbf{f_a}), w \neq null, u \neq null.$

$Flight\_(x0, x1, x2, x3, x4, x5, x6, x7, x8, \mathbf{f_a}) \vee Plane\_(x8, null, \mathbf{t_a}) \leftarrow$

$Flight\_(x0, x1, x2, x3, x4, x5, x6, x7, x8, \mathbf{t^\star}),\ not\ aux3\_(x8), x8 \neq null.$

$aux3\_(x8) \leftarrow Plane\_(x8, z, \mathbf{t^\star}),\ not\ Plane\_(x8, z, \mathbf{f_a}), x8 \neq null, z \neq null.$

$\leftarrow Planetype\_(x, y, z, w, \mathbf{t_a}), Planetype\_(x, y, z, w, \mathbf{f_a}).$

$\leftarrow Plane\_(x, y, \mathbf{t_a}), Plane\_(x, y, \mathbf{f_a}).$ □

We can notice that the repair program will import the whole database to DLV to evaluate programs. Instead, the MS and the RP methodologies generate programs that will import only a subset of the relations in the database. We illustrate this below.

For evaluation purposes we consider the following three conjunctive queries:

1. $Ans(x, y, z) \leftarrow Plane(x, y), Planetype(y, w, z, s)$, i.e. a conjunctive query with projection, but without constants, that asks for the id, type, and maximum capacity of passengers of planes.

2. $Ans(x, y) \leftarrow Passenger(1, w, v), Flying(1, y, x)$, i.e. a partially-ground conjunctive query with projections, that asks for the flight, and tag numbers for passenger with *PID* equals to 1.

3. $Ans \leftarrow Passenger(1, smith, 1)$, i.e. a boolean (or ground) conjunctive query.

For query $Ans(x, y, z) \leftarrow Plane(x, y), Planetype(y, w, z, s)$, the programs generated by the MS and the RP methodologies contain import sentences for predicates *Plane, Planetype, Flight,* and *Inspection*, only. This is because those are the relevant predicates to compute the query (cf. Dependency Graph in Figure 9.13). Therefore, DLV will import only the tuples stored in the relations that are relevant to compute the query, but not the whole set of relations.

For query $Ans(x,y) \leftarrow Passenger(1,w,v), Flying(1,y,x)$, the programs generated by RP and MS contain import sentences for predicates *Passenger* and *Flying*, only. Nevertheless, the MS program will import a more reduced amount of tuples, than the repair program generated by the RP methodology.

Program 9.5 is the MS program (without the import sentences) for the partially-ground query $Ans(x,y) \leftarrow Passenger(1,w,v), Flying(1,y,x)$.

Program 9.5

$Ans(x) \leftarrow magic\_Ans^f, Passenger\_(1,w,v,\mathbf{t}^{\star\star}), Flying\_(1,y,x,\mathbf{t}^{\star\star})$.

$Flying\_(x0,x1,x2,\mathbf{t}^{\star}) \leftarrow magic\_Flying\_^{bffb}(x0,\mathbf{t}^{\star}), Flying\_(x0,x1,x2,\mathbf{t_a})$.

$Flying\_(x0,x1,x2,\mathbf{t}^{\star}) \leftarrow magic\_Flying\_^{bffb}(x0,\mathbf{t}^{\star}), Flying(x0,x1,x2)$.

$Flying\_(x0,x1,x2,\mathbf{t}^{\star\star}) \leftarrow magic\_Flying\_^{bffb}(x0,\mathbf{t}^{\star\star}), Flying\_(x0,x1,x2,\mathbf{t}^{\star}),$

$\quad not\ Flying\_(x0,x1,x2,\mathbf{f_a})$.

$Passenger\_(x0,x1,x2,\mathbf{t}^{\star}) \leftarrow magic\_Passenger\_^{bffb}(x0,\mathbf{t}^{\star}), Passenger\_(x0,x1,x2,\mathbf{t_a})$.

$Passenger\_(x0,x1,x2,\mathbf{t}^{\star}) \leftarrow magic\_Passenger\_^{bffb}(x0,\mathbf{t}^{\star}), Passenger(x0,x1,x2)$.

$Passenger\_(x0,x1,x2,\mathbf{t}^{\star\star}) \leftarrow magic\_Passenger\_^{bffb}(x0,\mathbf{t}^{\star\star}), Passenger\_(x0,x1,x2,\mathbf{t}^{\star}),$

$\quad not\ Passenger\_(x0,x1,x2,\mathbf{f_a})$.

$Passenger\_(x,y,w,\mathbf{f_a}) \vee Passenger\_(x,z,t,\mathbf{f_a}) \leftarrow magic\_Passenger\_^{bffb}(x,\mathbf{f_a}),$

$\quad magic\_Passenger\_^{bffb}(x,\mathbf{f_a}),\ Passenger\_(x,y,w,\mathbf{t}^{\star}), Passenger\_(x,z,t,\mathbf{t}^{\star}),$

$\quad y \neq z, x \neq null, z \neq null, y \neq null$.

$Passenger\_(x,y,w,\mathbf{f_a}) \vee Passenger\_(x,z,s,\mathbf{f_a}) \leftarrow magic\_Passenger\_^{bffb}(x,\mathbf{f_a}),$

$\quad magic\_Passenger\_^{bffb}(x,\mathbf{f_a}), Passenger\_(x,y,w,\mathbf{t}^{\star}), Passenger\_(x,z,s,\mathbf{t}^{\star}),$

$\quad w \neq s, x \neq null, s \neq null, w \neq null$.

$magic\_Passenger\_^{bffb}(1,\mathbf{t}^{\star\star}) \leftarrow magic\_Ans^f$.

$magic\_Flying\_^{bffb}(1, \mathbf{t}^{\star\star}) \leftarrow magic\_Ans^{f}.$

$magic\_Flying\_^{bffb}(x0, \mathbf{t_a}) \leftarrow magic\_Flying\_^{bffb}(x0, \mathbf{t}^{\star}).$

$magic\_Flying\_^{bffb}(x0, \mathbf{t}^{\star}) \leftarrow magic\_Flying\_^{bffb}(x0, \mathbf{t}^{\star\star}).$

$magic\_Flying\_^{bffb}(x0, \mathbf{f_a}) \leftarrow magic\_Flying\_^{bffb}(x0, \mathbf{t}^{\star\star}).$

$magic\_Passenger\_^{bffb}(x0, \mathbf{t_a}) \leftarrow magic\_Passenger\_^{bffb}(x0, \mathbf{t}^{\star}).$

$magic\_Passenger\_^{bffb}(x0, \mathbf{t}^{\star}) \leftarrow magic\_Passenger\_^{bffb}(x0, \mathbf{t}^{\star\star}).$

$magic\_Passenger\_^{bffb}(x0, \mathbf{f_a}) \leftarrow magic\_Passenger\_^{bffb}(x0, \mathbf{t}^{\star\star}).$

$magic\_Passenger\_^{bffb}(x, \mathbf{f_a}) \leftarrow magic\_Passenger\_^{bffb}(x, \mathbf{f_a}).$

$magic\_Passenger\_^{bffb}(x, \mathbf{t}^{\star}) \leftarrow magic\_Passenger\_^{bffb}(x, \mathbf{f_a}).$

$magic\_Passenger\_^{bffb}(x, \mathbf{t}^{\star}) \leftarrow magic\_Passenger\_^{bffb}(x, \mathbf{f_a}).$

$magic\_Ans^{f}.$ □

We can see that the rules in the MS program involving database atoms are:

$$Flying\_(x0, x1, x2, \mathbf{t}^{\star}) \leftarrow magic\_Flying\_^{bffb}(x0, \mathbf{t}^{\star}), Flying(x0, x1, x2).$$

$$Passenger\_(x0, x1, x2, \mathbf{t}^{\star}) \leftarrow magic\_Passenger\_^{bffb}(x0, \mathbf{t}^{\star}), Passenger(x0, x1, x2).$$

In particular, for the first rule, in order to generate the atom $Flying\_(x0, x1, x2, \mathbf{t}^{\star})$ in the program, a database fact of the form $Flying(\bar{c})$ has to be true in the database, and the magic atom $magic\_Flying\_^{bffb}(x0, \mathbf{t}^{\star})$ has to be true in the program. This magic atom says that the first argument, i.e. variable $x0$, is bound in the predicate $Flying\_$ (also the last argument, which is the constant $\mathbf{t}^{\star}$). This means that in the evaluation of the MS program, only a subset of the tuples from relation $Flying$ will be used in the computation of the program. Those, that in the first attribute have the value taken by variable $x0$ in the magic atom $magic\_Flying\_^{bffb}(x0, \mathbf{t}^{\star})$. This magic atom is generated by the rule:

$$\mathit{magic\_Flying}^{bffb}(x0, \mathbf{t}^{\star}) \leftarrow \mathit{magic\_Flying}^{bffb}(x0, \mathbf{t}^{\star\star}).$$

The atom $\mathit{magic\_Flying}^{bffb}(x0, \mathbf{t}^{\star\star})$ is generated by the rule:

$$\mathit{magic\_Passenger\_}^{bffb}(1, \mathbf{t}^{\star\star}) \leftarrow \mathit{magic\_Ans}^{f}.$$

Therefore, since the magic seed atom $\mathit{magic\_Ans}^{f}$ is true in the program (cf. Chapter 5), the variable $x0$ will take the value 1.

The same analysis can be done for relation *Passenger*. For this relation, also the tuples with value 1 in the first attribute will be used in the computation of the MS program. This attribute is *PID* in both relations. Therefore, the import sentences in the MS program are:

$$\#\mathit{import}(\mathit{airport}, \text{“}\mathit{db2inst2}\text{”}, \text{“}\mathit{db2inst2}\text{”}, \text{“}\mathit{SELECT\ DISTINCT} *$$
$$\mathit{FROM\ Passenger\ WHERE\ PID} = \mathit{1}\text{”}, \mathit{Passenger}).$$
$$\#\mathit{import}(\mathit{airport}, \text{“}\mathit{db2inst2}\text{”}, \text{“}\mathit{db2inst2}\text{”}, \text{“}\mathit{SELECT\ DISTINCT} *$$
$$\mathit{FROM\ Flying\ WHERE\ PID} = \mathit{1}\text{”}, \mathit{Flying}). \tag{9.2}$$

While the import sentences in the repair program generated by the RP methodology are:

$$\#\mathit{import}(\mathit{airport}, \text{“}\mathit{db2inst2}\text{”}, \text{“}\mathit{db2inst2}\text{”}, \text{“}\mathit{SELECT\ DISTINCT} *$$
$$\mathit{FROM\ Passenger}\text{”}, \mathit{Passenger}).$$

$$\#import(airport, \text{“}db2inst2\text{”}, \text{“}db2inst2\text{”}, \text{“}SELECT\ DISTINCT\ *$$
$$FROM\ Flying\text{”}, Flying). \tag{9.3}$$

For query $Ans \leftarrow Passenger(1, smith, 1)$, both programs generated by MS and RP contain only an import sentence for relation *Passenger*. However, the MS program will import the tuples of *Passenger* that have *PID* equals to 1, or *PNAME* equals to *smith*, or *PHONE* equals to 1. Because MS will push-down these constants into the rules for the predicate $Passenger\_$ in the MS program. The import sentence in the MS program is:

$$\#import(airport, \text{“}db2inst2\text{”}, \text{“}db2inst2\text{”}, \text{“}SELECT\ DISTINCT\ *$$
$$FROM\ Passenger\ WHERE\ PID = 1$$
$$OR\ PNAME = \text{‘}smith\text{'}\ OR\ PHONE = 1\text{”}, Passenger).$$

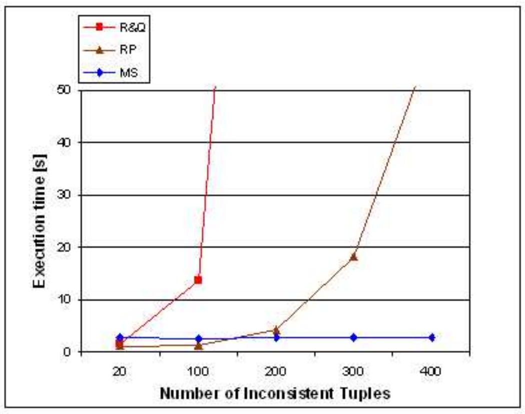
The queries were evaluated on database instances D_1 and D_2 with 3200, and 6400 tuples, respectively. Each database had n inconsistent tuples wrt the primary keys of relations *Passenger*, *Plane*, and *Flight*. We considered values for n in the range of 20-400. Also, the inconsistent tuples in relation *Flight* violated the RIC $\forall x0x1x2x3x4x5x6x7x8\ (Flight(x0, x1, x2,\ x3, x4, x5, x6, x7, x8) \rightarrow \exists z\ Plane(x8, z))$, and the inconsistent tuples in relation *Plane* violated the RIC $\forall zx(Plane(z, x) \rightarrow \exists yuw(Planetype(x, y, u, w))$.

9.4.2 Experimental Results

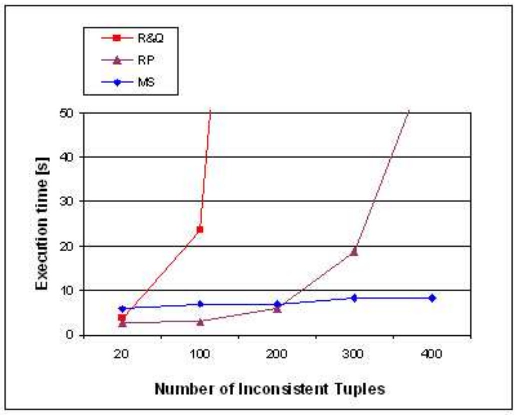
Figures 9.14, 9.15, and 9.17 show the execution times of the three queries on database instances D_1 and D_2, respectively. In the charts, *R&Q* denotes the straightforward evaluation of programs.

Figure 9.14 shows the running time for the conjunctive query $Ans(x, y, z) \leftarrow$

$Plane(x, y), Planetype(y, w, z, s)$. First, we can see that the optimized methodologies in *ConsEx* are faster to compute answers than the straightforward evaluation of programs. As an illustration, for $n = 200$ the MS and the RP methodologies return answers in less than ten seconds, while the straightforward evaluation of programs returns answers after one minute (in both database instances).



Database Instance D_1



Database Instance D_2

Figure 9.14: Running Time for the Conjunctive Query with Projections

Second, we can see that the execution times of the MS methodology are almost invariant, while the execution times of the RP methodology grow-up when the database contains more inconsistent tuples. For instance, for n greater than 200 the execution times of RP start growing up considerably. Notice also that for n less that 200 the RP methodology returns answers a bit faster than MS. This happen because the instantiation of the MS program takes more time in DLV, than the instantiation of the program generated by the RP methodology. This is because the MS program has more rules than the RP program (cf. Chapter 5). The instantiation time is almost invariant for the MS program, but for n less than 200 it is greater than the time needed by the program generated by the RP methodology. As a consequence, for n less than 200 RP returns answers faster than MS. However, since MS performs a partial computation of the stable models, and simulates a top-down evaluation of queries, the time that MS spends in the computation of queries is slower than the time consumed by the RP methodology, which does not perform any of those additional optimizations. This is more notice for n greater than 200. In particular, when $n = 400$ MS returns answers in less than ten seconds (in both database instances), while RP returns answers after one minute.

Moreover, the scalability of the optimized methods to compute consistent answers is good. As an illustration, the database instance D_2 doubles the amount of tuples in database instance D_1, and MS still returns consistent answers in less than ten seconds.

Figure 9.15 shows the execution time for the partially-ground query $Ans(x, y) \leftarrow Passenger(1, w, v), Flying(1, y, x)$.

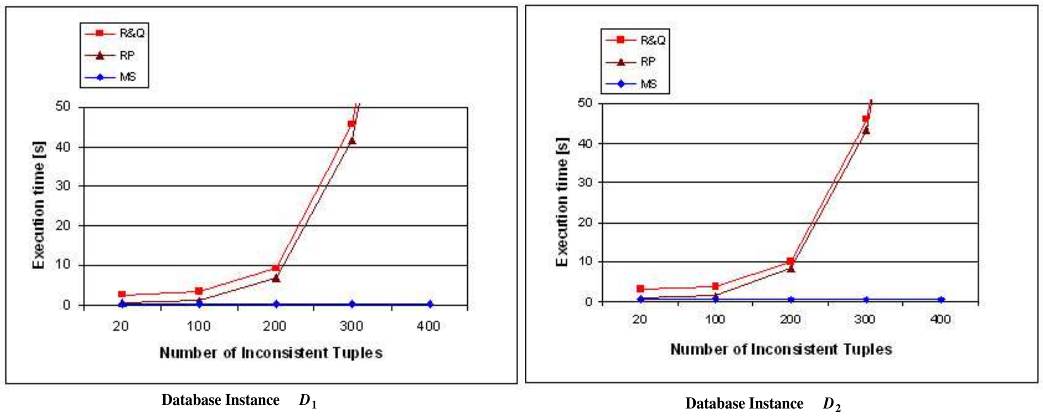


Figure 9.15: Running Time for the Partially-Ground Conjunctive Query

We can see that, as expected, MS is much better to compute the partially-ground conjunctive query than the other methods (in both database instances). This happens because, since the query has constants, less tuples are imported to DLV and, as an immediate consequence the instantiation of the MS program is reduced. Moreover, MS pushes-down the query constants and the query is evaluated in a simulated top-down way, which produces a better performance.

Moreover, for this query the difference between the MS and the RP methodology is more considerable in both database instances. As an illustration, in Figure 9.14, in both database instances, the MS curve intersects the RP curve at (more less) $n = 200$, but in Figure 9.15 this happens before, when $n = 100$ (more less). After that point, the execution times of the RP methodology grow-up considerably.

MS produces gain in the execution times of partially-ground queries even if the constants in the query are not considered when importing relations to DLV. For instance, if for query $Ans(x, y) \leftarrow Passenger(1, w, v), Flying(1, y, x)$ we use the import sentences in 9.3 instead of the import sentences in 9.2, MS still has a good performance computing the query. We proved it in the largest database instance D_2. Figure 9.16 shows the execution times for the partially-ground conjunctive query computed with the MS program with the import sentences in 9.3 (MS in the chart), and the import sentences in 9.2 (MS-NF in the chart).

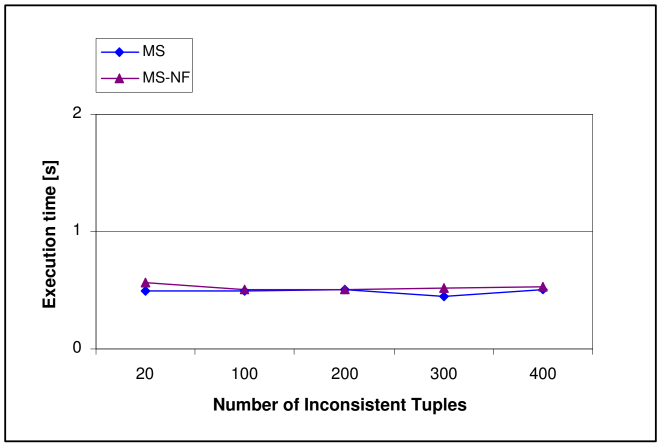
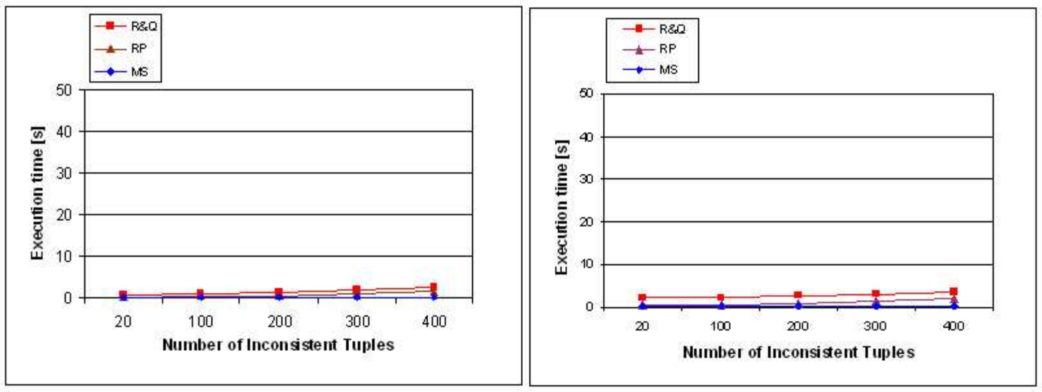


Figure 9.16: Running Times of MS

Figure 9.17 shows the execution time for query $Ans \leftarrow Passenger(1, smith, 1)$. Here, we can appreciate that, as expected, the methods implemented in *ConsEx* are faster to compute answers than the direct evaluation of programs. Moreover, MS returns answers faster than the RP methodology, but the difference in the running times of *MS* and RP is not considerable. Also, the scalability of the methods is good, since the running times of both methods are very similar in the database instance D_2, which has the double of tuples of D_1.



Database Instance D_1 Database Instance D_2

Figure 9.17: Running Time for the Boolean Query

Given the experimental results, we conclude that any of the methods implemented in *ConsEx* is faster to compute queries than the straightforward evaluation of programs. Moreover, for conjunctive queries with projections, MS computes answers much faster than the RP methodology and, therefore, we achieve a better performance in query processing by using *MS*. Furthermore, we could use the RP methodology instead of MS to evaluate boolean conjunctive queries with one database atom, since for this kind of queries, both methodologies have similar performance (cf. Figure 9.17). This may be important, since if we use the RP methodology, we do not have to generate the extra MS program.

In this analysis we did not consider the time that the system takes in the generation of programs. Basically because since the system does not bring any data from the database to the main memory, the generation of programs is considerably fast.

9.5 Summary

In this chapter we described *ConsEx*, which is the first logic programming-based system for CQA over inconsistent databases. The system implements our MS methodology, and the RP methodology to compute consistent answers to FO queries (cf. Chapter 5). In *ConsEx* only the relevant data to compute queries is imported to DLV. In this manner, the flow of data between *ConsEx* and the database is reduced.

We performed experiments in databases instances of 3200 and 6400 tuples, with different levels of inconsistency. We consider three examples of conjunctive queries. It was shown that the optimized methodologies are faster to compute queries than the straightforward evaluation of programs. In particular, for conjunctive queries with projection, MS computes answers much faster than the RP methodology. Hence, MS is a good alternative to optimize CQA over inconsistent databases. For boolean queries with one database atom, the RP methodology has good performance, and it can be used instead of MS to evaluate queries. In this case, the generation of the extra, new rewritten MS program, is avoided.

The optimized methods to compute consistent answers implemented in *ConsEx* have a good scalability in the databases instances we considered. It is a future work, to perform more experiments considering larger databases, and other classes of queries like disjunctive queries. Also, we leave as feature work the implementation of aggregate queries in *ConsEx*.

Based in the experiments performed, we conclude that CQA based on logic programs is viable, and can be efficiently implemented. Especially because it is not expected that databases contain enormous amounts of inconsistent tuples.

Chapter 10

Conclusions

In this thesis we were concerned with finding ways to improve the logic approach for CQA. We claimed that even though we cannot reduce the intrinsic complexity of CQA over inconsistent databases, which in the worst case is Π_2^P-complete in data complexity [28], we can optimize CQA based on logic programs.

We started by defining structural optimizations to repair programs. Essentially, annotations on database facts and auxiliary predicates were eliminated. The elimination of the annotations on database facts permit us to import facts directly to the DLV system, where programs are evaluated. This improves both the generation of repair programs, and their evaluation.

We also presented a methodology to generate only the needed program constraints for repair programs. Program constraints avoid the generation of incoherent models. Originally they were defined for every database predicate, but it was shown that only a subset of the program constraints are needed in repair programs. The elimination of these rules is important because it eliminates unnecessary model checking in a reasoning system. In addition, important classes of ICs were identified for which repair programs can be specified without program constraints at all. It becomes relevant when magic sets techniques are applied with the DLV system, which implements magic sets for programs without program constraints.

Therefore, with the structural optimizations we generated new programs that have less rules and annotations than the original ones. Moreover, we showed that the repair programs (without considering the program constraints) are locally stratified, which was important to prove that the MS methodology we defined, for disjunctive repair programs with program constraints, is sound and complete.

The evaluation of programs in reasoning systems has also been optimized. We defined a suitable MS methodology for our repair programs with program constraints. MS techniques allow the focalization on part of programs and facts that are relevant to answer a query. The methodology works as follows: The set of program constraints (if any) is separated from the rest of the rules in the repair program, then the MS technique is applied to the resulting program. Program constraints are added to the rewritten program generated by the MS technique.

It is important to mention that the methodology works for our repair programs and program constraints, but not necessarily for general constraint rules, i.e. rules of the form $\leftarrow C(\overline{x})$, where $C(\overline{x})$ is a conjunction of literals (a positive or negated atom). For this kind of rules in disjunctive programs is known that MS is sound but incomplete [47].

We also presented a way to encode program constraints of repair programs (if any) in order to apply MS in the DLV system, which does not support MS for programs with constraints. DLV applies MS internally, without returning the rewritten program. This implies that it is not possible to add the program constraints later to the magic program. Nevertheless, this extra processing of programs is avoided in the "Consistency Extractor System", which implements our MS methodology for repair programs. However we decided to present the methodology of encoding program constraints into programs because it is interesting in itself.

Furthermore, we presented a different optimization methodology, which also captures the relevant database predicates to compute a specific query. However, relevant predicates are captured by analyzing the relationship between predicates in the ICs and the query predicates, which is captured by the dependency graph (cf. Definition 2.1).

The relevant predicates are used to generate repair programs for a subset of the database relations. As a matter of fact, only the rules, and more important the base data, for predicates that are relevant to compute a query are kept in programs. In particular, this produce a reduction of the flow of data between the database system and the reasoning system where programs are evaluated.

Moreover, we presented a logic programming specification to compute consistent answers to aggregate queries with both *scalar* functions, and *group-by* statements wrt FDs. For the former queries, the notion of consistent answers we considered is the same presented in [4]. For the latter, we introduced a notion of consistent answer, which is based in the notion of consistent answers to scalar queries. The logic programming specification is restricted to FDs. Nevertheless, it should be possible to apply this approach straightforwardly to RIC-acyclic sets of ICs.

The logic programming specification is based in the formalism supported by the DLV system. The current version of DLV implements five aggregation functions. However, there are technical difficulties when aggregates are defined over atoms appearing in the head of disjunctive rules. In theses cases problems arise during the grounding process which is executed before the computation of the stable models. Specifically, the variable that holds the aggregate value can become unbound when the ground version of the program is computed, and the DLV system would have to compute all possible values to bind it. For some functions, such as `max` and `min` this problem can be solved by adding extra atoms into the aggregate rule to bind the aggregate variable, but for other functions such as `sum` that is not possible, and grounding could become very difficult. Nevertheless, it is important to remark that this is only a technical difficulty, that should be solved in a future release of the DLV system.

In this thesis we also analyzed the use of the WFS of programs to CQA. This semantics has lower data complexity than the stable models semantics of programs, and therefore it becomes a good alternative to compute consistent answers. Actually, the WFI of a program can be computed in polynomial time (data complexity).

It was proven that for *interaction-free* sets of ICs (cf. Definition 7.6), and conjunctive queries without existential quantifiers, the core answers, i.e. the answers obtaining from the intersection of all the stable models of program $\Pi(D, IC, \mathcal{Q})$, coincide with the well-founded answers, obtained from W^+ of the WFI of $\Pi(D, IC, \mathcal{Q})$. This is relevant because, since the WFI of a program is computed in polynomial time, we can compute consistent answer to this kind of queries in polynomial time. This result extends preliminary results presented in [3], which hold for a set of ICs containing FDs and unary ICs only.

In addition, it was shown that when the set of ICs is restricted to FDs and considering at most one FD or key dependency per relation, it is possible to use W^u of the WFI of programs for CQA. In fact, by using both W^+ and W^u, we can compute all the consistent answers for a restricted set of conjunctive queries with existential quantifiers. In order to use W^u of the WFI of programs, it is necessary to rewrite queries. The rewriting we used coincides with the rewriting method presented in [39], that works for primary key constraints, and a more general class of conjunctive queries.

We also use the WFS of programs as first step to compute consistent answers to ground disjunctive queries wrt *interaction-free* sets of ICs. In this manner, the computation of stable models is left as a second option if needed. Moreover, it was shown that if the WFI of a program is used as a unique way to compute consistent answers, for positive Datalog queries, i.e. conjunctive or disjunctive queries with projection but without negation, we retrieve only a subset of the consistent answers. Nevertheless, we have a polynomial time algorithm for CQA.

When programs are HCF [9], the well-founded answers can be computed in the XSB system [71]. Also, for sets of *interaction-free* ICs, the well-founded answers can be computed by using the deterministic set of programs, which is efficiently computed in the DLV system [19]. This is because, for *interaction-free* sets of ICs, the deterministic set and W^+ of the WFI of $\Pi(D, IC, \mathcal{Q})$, coincide.

Table 10.1 summarizes the optimizations to CQA presented in this thesis.

Optimization	Contribution
Structural Changes to Programs	Smaller programs that are easier to evaluate
MS Methodology	Focalization on relevant parts of programs
Methodology to encode denial rules	Use of MS as implemented in DLV
Analysis of WFS	Identification of ICs and queries for which the WFS retrieves all (and subsets) of the consistent answers
Logic programs to compute aggregate queries	Extension of logic programs to compute aggregate queries, by using the capabilities of DLV to compute aggregates
ConsEx	The first optimized logic programming-based implementation to compute consistent answers to FO queries

Table 10.1: Summary of Optimizations and Contributions

In this thesis we also developed a repair semantics for Multidimensional Databases. We provided a suitable notion of repair for multidimensional dimension instances wrt partitioning constraints (MPC). A repair of a multidimensional dimension instance satisfies the MPC, however is not homogeneous. This is because some roll-up functions on it are not total. Moreover, the summarizability property cannot be reestablished in the repairs. Nevertheless, we could recover summarizability by introducing "dummy" elements in some categories in the repairs as in [65]. It is a future work to analyze if this way of recovering summarizability can be used to repair MDWs and compute consistent answers.

The analysis of consistency in MDBs is a preliminary study that we will extend to heterogeneous dimension schemas [48], where the situation could be a bit different; mainly because of the different dimension constraints.

For future research we leave the improvement of the definition of dimension repairs by using knowledge from equality constraints [48]. These constraints impose the existence of certain distinguished elements in the categories. Thus, we could require that repairs satisfy those constraints to get more meaningful repairs. Also, it is a future work to develop a methodology for computing repairs (if necessary, because this should be avoided whenever possible due to its complexity) and consistent answers.

Moreover we are interested in analyzing consistent query answering wrt aggregation constraints [73]. Also, we will analyze the idea of translating dimension instances into sets of normalized relational tables before repairing them. This by following the idea presented in [77], where a project-join dependency is applied to a relation prior to repairing wrt FDs. In this way, they obtain more meaningful repairs wrt FDs.

Finally, we developed the "Consistency Extractor System" (*ConsEx*), the first optimized logic programming-based implementation to compute consistent answers to FO queries, from stand alone relational databases. It is a future work to extend *ConsEx* to compute consistent answers to aggregate queries.

ConsEx implements our MS methodology to disjunctive repair programs with program constraints, and the RP methodology, which generates programs considering the relevant predicates, i.e. predicates that are involved in the computation of queries (cf. Chapter 5).

We reported experimental results performed on database instances with different levels of inconsistency (cf. Chapter 9). The results showed that the optimized methods implemented in *ConsEx* are faster to compute consistent answers to queries than the straightforward evaluation of programs. Moreover, the MS methodology is faster returning answers to conjunctive queries than the RP methodology. Nevertheless, the latter methodology could be used, instead of MS, to evaluate boolean conjunctive queries with one database atom, since for this kind of queries, both methodologies have similar performance.

Based in the experiments performed, we conclude that CQA based on logic programs is viable, and can be efficiently implemented. Especially because it is not expected that databases contain enormous amounts of inconsistent tuples.

Bibliography

[1] Abiteboul, S., Hull, R. and Vianu, V. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Arenas, M., Bertossi, L. and Chomicki, J. Consistent Query Answers in Inconsistent Databases. In *Proc. 18th ACM Symposium on Principles of Database Systems (PODS 99)*, ACM Press, 1999, pp. 68–79.

[3] Arenas, M., Bertossi, L. and Chomicki, L. Answer Sets for Consistent Query Answering in Inconsistent Databases. *Theory and Practice of Logic Programming*, 2003, 3(4-5): 393–424.

[4] Arenas, M., Bertossi, L. and Chomicki, J. Scalar Aggregation in FD-Inconsistent Databases. In *Proc. International Conference on Database Theory (ICDT 01)*, Springer LNCS 1973, 2001, pp. 39–53.

[5] Arenas, M., Bertossi, L., Chomicki, J., He, X., Raghavan, V. and Spinrad, J. Scalar Aggregation in Inconsistent Databases. *Theoretical Computer Science*, 2003, 296(3):405–434.

[6] Barcelo, P. and Bertossi, L. Logic Programs for Querying Inconsistent Databases. In *Proc. 5th International Symposium on Practical Aspects of Declarative Languages (PADL 03)*, Springer LNCS 2562, 2003, pp. 208–222.

[7] Barcelo, P., Bertossi, L. and Bravo, L. Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. Chapter in book *Semantics of Databases*, Springer LNCS 2582, 2003, pp. 1–27.

[8] Beeri, C. and Ramakrishnan, R. On the Power of Magic. In *Proc. 6th ACM Symposium on Principles of Database Systems (PODS 87)*, ACM Press, 1987, pp. 269–284.

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

[10] Bertossi, L. and Chomicki, J. Query Answering in Inconsistent Databases. Chapter in book *Logics for Emerging Applications of Databases*, J. Chomicki, G. Saake and R. van der Meyden (eds.), Springer, 2003, pp. 43–83.

[11] Bravo, L. and Bertossi, L. Logic Programs for Consistently Querying Data Sources. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI 03)*, Morgan Kaufmann, 2003, pp. 10–15.

[12] Bravo, L. and Bertossi, L. Consistent Query Answering under Inclusion Dependencies. In *14th Annual IBM Centers for Advanced Studies Conference (CASCON 2004)*, 2004, pp. 202–216.

[13] Bertossi, L. and Bravo, L. Query Answering in Peer-to-Peer Data Exchange Systems. In *Current Trends in Database Technology*. Springer LNCS 3268, 2004, pp. 478–485.

[14] Bertossi, L., Bravo, L., Franconi, E. and Lopatenko, A. Complexity and Approximation of Fixing Numerical Attributes in Databases Under Integrity Constraints. In *Proc. of the Databases Programming Languages Conference (DBPL 05)*, Springer LNCS 3774, 2005, pp. 262–278.

[15] Bravo, L. and Bertossi, L. Deductive Databases for Computing Certain and Consistent Answers from Mediated Data Integration Systems. *Journal of Applied Logic*, 2005, 3(2):329–367.

[16] Bravo, L. and Bertossi, L. Semantically Correct Query Answers in the Presence of Null Values. In *Pre-Proc. EDBT WS on Inconsistency and Incompleteness in Databases (IIDB 06)*, J. Chomicki and J. Wijsen (eds.), 2006, pp. 33–47.

[17] Cali, A., Lembo, D. and Rosati, R. On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In *Proc. Symposium on Principles of Database Systems (PODS 03)*, ACM Press, 2003, pp. 260–271.

[18] Cali, A., Lembo, D. and Rosati, R. Query Rewriting and Answering Under Constraints in Data Integration Systems. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI 03)*, Morgan Kaufmann, 2003, pp. 16–21.

[19] Calimeri, F., Faber, W., Leone, N. and Pfeifer, G. Pruning Operators for Disjunctive Logic Programming Systems. *Fundamenta Informaticae*, 2006, 71(2-3):183-214.

[20] Cao, T., Pontelli, E. and Elkabani, I. On Logic Programming with Aggregates. NMSU Technical Report, NMSU-CS-2005-005, 2005.

[21] Caniupan, M. and Bertossi, L. Optimizing Repair Programs for Consistent Query Answering. In *Proc. 25th International Conference of the Chilean Computer Science Society (SCCC 2005)*, IEEE Computer Society Press, 2005, pp. 3–12.

[22] Celle, A. and Bertossi, L. Querying Inconsistent Databases: Algorithms and Implementation. In *Computational Logic - CL 2000, Stream: International Conference on Rules and Objects in Databases (DOOD 00)*, Springer LNAI 1861, 2000, pp. 942–956.

[23] Ceri, S., Gottlob, G. and Tanca, L. *Logic Programming and Databases*. Springer-Verlag, 1990.

[24] Chaudhuri, S. and Dayal, U. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 1997, 26(1):65–74.

[25] Chen, W., Swift, T. and Warren, D. *Efficient Top-Down Computation of Queries under the Well-Founded Semantics. Journal of Logic Programming*, 1995, 24(3):161–199.

[26] Chomicki, J. and Marcinkowski, J. On the Computational Complexity of Consistent Query Answers. CoRR paper cs.DB/0204010, 2002.

[27] Chomicki, J. and Marcinkowski, J. Minimal-Change Integrity Maintenance Using Tuple Deletions. *Information and Computation*, 2005, 197(1-2):90–121.

[28] Chomicki, J. and Marcinkowski, J. On the Computational Complexity of Minimal-Change Integrity Maintenance in Relational Databases. In *Inconsistency Tolerance*, Springer LNCS 3300, State of the Art Survey Series, 2004, pp. 119–150.

[29] Cumbo, C., Faber, W., Greco, G. and Leone, N. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In *Proc. 20th International Conference on Logic Programming (ICLP 04)*, Springer LNCS 3132, 2004, pp. 371–385.

[30] Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. Complexity And Expressive Power of Logic Programming. *ACM Computer Surveys*, 2001, 33(3):374–425.

[31] Dell'Armi, T., Faber, W., Ielpa, G., Leone, N. and Pfeifer, G. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI 03)*, Morgan Kaufmann, 2003, pp. 847–852.

[32] Dell'Armi, T., Faber, W., Ielpa, G., Leone, N. and Pfeifer, G. Aggregate Functions in DLV. In *Proc. Answer Set Programming: Advances in Theory and Implementation*, Marina de Vos and Alessandro Provetti, 2003, pp. 274–288.

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

[34] Eiter, T., Fink, M., Greco, G. and Lembo, D. Efficient Evaluation of Logic Programs for Querying Data Integration Systems. In *Proc. 19th International Conference on Logic Programming (ICLP 03)*, Springer LNCS 2916, 2003, pp. 163–177.

[35] Faber, W., Greco, G. and Leone, N. Magic Sets and their Application to Data Integration. In *Proc. International Conference on Database Theory (ICDT 05)*, Springer LNCS 3363, 2005, pp. 306–320.

[36] Faber, W., Leone, N. and Pfeifer, G. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Proc. 9th European Conference on Artificial Intelligence (JELIA 2004)*, Springer LNCS 3229, 2004, pp. 200–212.

[37] Faber, W. Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates. In *Logic Programming and Nonmonotonic Reasoning, 8th International Conference (LPNMR'05)*, Springer Verlag, 2005, pp. 40–52.

[38] Franconi, E., Laureti-Palma, A., Leone, L., Perri, S. and Scarcello, F. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In *Proc. of the Artificial Intelligence on Logic for Programming (LPAR 01)*, Springer Verlag (3-540-42957-3), 2001, pp. 561–578.

[39] Fuxman, A. and Miller, R. First-Order Query Rewriting for Inconsistent Databases. In *Proc. International Conference on Database Theory (ICDT 05)*, Springer LNCS 3363, 2004, pp. 337–354.

[40] Garcia-Molina, H., Labio, W. J. and Yang, J. Expiring Data in a Warehouse. In *Proc. 24th International Conference on Very Large Data Bases (VLDB 98)*, 1998, pp. 500–511.

[41] Gupta, H. and Mumick, I. S. Selection of Views to Materialize under a Maintenance Cost Constraint. In *Proc. 7th International Conference on Database Theory (ICDT 99)*, Springer LNCS 1540, 1999, pp. 453–470.

[42] Gelfond, M. and Leone, N. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 2002, 138(1-2):3–38.

[43] Gelfond, M. and Lifschitz, V. The Stable Model Semantics for Logic Programming. In *Logic Programming, Proc. 5th International Conference and Symposium (ICLP/SLP 88)*, MIT Press, 1988, pp. 1070–1080.

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

[45] Greco, S. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. In *IEEE Transactions on Knowledge and Data Engineering*, 2003, 15(2):368–385.

[46] Greco, G., Greco, S. and Zumpano, E. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2003, 15(6):1389–1408.

[47] Greco, G., Greco, S., Trubtsyna, I. and Zumpano, E. Optimization of Bound Disjunctive Queries with Constraints. *Theory and Practice of Logic Programming*, 2005, 5(6):713–745.

[48] Hurtado, C. Structurally Heterogeneous Olap Dimensions. *Doctoral Thesis, Computer Science Depto, University of Toronto*, 2002.

[49] Hurtado, C., Gutierrez, C. and Mendelzon, A. Capturing Summarizability with Integrity Constraints in OLAP. *ACM Transactions on Database Systems*, 2005, 30(3):854–886.

[50] Hurtado, C., Mendelzon, A. and Vaisman, A. Updating OLAP Dimensions. In *Proc. 2nd IEEE-DOLAP Workshop, Kansas City, Missouri, USA*, 1999, pp. 60–66.

[51] Hurtado, C., Mendelzon, A. and Vaisman, A. Maintaining Data Cubes under Dimension Updates. In *Proc. 15th IEEE-ICDE Conference, Sydney, Australia*, 1999, pp. 346–357.

[52] Johnson, C.A. Top-down Query Processing in First Order Deductive Databases under the WFS. In *Proc. 12th International Symposium on Foundations of Intelligent Systems*, Springer-Verlag, 2000, pp. 377–388.

[53] Lakshmanan, L., Ng, R., Xing Wang, C., Zhou. X. and Johnson, T. The Generalized MDL Approach for Summarization. In *Proc. 28th Int. Conf. Very Large Data Bases, (VLDB 02), Hong Kong, China*, 2002, pp. 766–777.

[54] Lenzerini, M. Data Integration: A Theoretical Perspective. In *Proc. ACM Symposium on Principles of Database Systems (PODS 02)*, ACM Press, 2002, pp. 233–246.

[55] Lembo, D., Lenzerini, M. and Rosati, R. Source Inconsistency and Incompleteness in Data Integration. In *Proc. Workshop on Knowledge Representation Meets Databases (KRDB 02)*, CEUR Electronic Workshop Proceedings, http://ceur-ws.org/Vol-54/, 2002.

[56] Letz, C., Henn, E. T. and Vossen, G. Consistency in Data Warehouse Dimensions. In *Proc. International Database Engineering and Applications Symposium, (IDEAS'02), IEEE Press (0-7695-1638-6)*, 2002, pp. 224–232.

[57] Levy, A. Logic-Based Techniques in Data Integration. Chapter in *Logic Based Artificial Intelligence*, J. Minker (ed.), Kluwer Publishers, 2000.

[58] 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 Staniszkis, W. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data In *Proc. International Conference on Management of Data (SIGMOD '05)*, ACM Press, 2005, pp. 915–917.

[59] Leone, N., Scarcello, F. and Subrahmanian, V.S. Optimal Models of Disjunctive Logic Programs: Semantics, Complexity, and Computation. *IEEE Transactions on Knowledge and Data Engineering*, 2004, 16(4):487–503.

[60] Leone, N., Rullo, P. and Scarcello, F. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. *Information and Computation*, 1997, 135(2):69–112.

[61] 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*, 2006, 7(3):499–562.

[62] Lifschitz, V. Circumscription. *Handbook of Logic in AI and Logic Programming, Oxford University Press*, 1994, 3:298–352.

[63] Lifschitz, V. and Turner, H. Splitting a Logic Program. In *Proc. International Conference on Logic Programming (ICLP 94)*, MIT Press, 1994, pp. 23–37.

[64] Lloyd, J.W. *Foundations of Logic Programming.* Second ed., Springer-Verlag, 1987.

[65] Pedersen, T., Jensen, C. and Dyreson, C. Extending Practical Pre-Aggregation in On-Line Analytical Processing. In *Proc. 25th Int. Conf. Very Large Data Bases, VLDB, Edinburgh, Scotland*, 1999, pp. 663–674.

[66] Przymusinski, T.C. On the Declarative Semantics of Deductive Databases and Logic Programs. *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publishers Inc., 1988, 193–216.

[67] Przymusinski, T.C. Well-Founded Semantics Coincides with Three-Valued Stable Semantics. *Fundamenta Informaticae*, IOS Press, 1990, 13(4):445–463.

[68] Przymusinski, T.C. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 1991, 9(3-4):401–424.

[69] Reiter, R. Towards a Logical Reconstruction of Relational Database Theory. In *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos, J.W. Schmidt (eds.), Springer, 1984, pp. 191–233.

[70] Ross, K. Modular Stratification and Magic Sets for Datalog Programs with Negation. *Journal of the ACM*, 1994, 41(6):1216–1266.

[71] Sagonas, K.F., Swift, T. and Warren, D.S. XSB as an Efficient Deductive Database Engine. In *Proc. International Conference on Management of Data (SIGMOD 94)*, ACM Press, 1994, pp. 442-453.

[72] Schlesinger, L. and Lehner, W. Extending Data Warehouses by Semi-Consistent Views. In *Proc. 4th International Workshop of Design and Management of Data Warehouses (DMDW 2002)*, CEUR Workshop Proceedings, 2002, pp. 43–51.

[73] Ross, K., Srivastava, D., Stuckey, P., and Sudarshan, S. Foundations of Aggregation Constraints. *Theoretical Computer Science*, 1998, 193(1-2):149–179.

[74] Theodoratos, D. and Bouzeghoub, M. A General Framework for the View Selection Problem for Data Warehouse Design and Evolution. In *Proc. 3rd ACM International Workshop on Data warehousing and OLAP*, ACM Press, 2000, pp. 1–8.

[75] Van Gelder, A., Ross, K.A. and Schlipf, J.S. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *Proc. Symposium on Principles of Database Systems (PODS 88)*, ACM Press, 1988, pp. 221–230.

[76] Wijsen, J. Condensed Representation of Database Repairs for Consistent Query Answering. In *Proc. 9th International Conference on Database Theory (ICDT 03)*, Springer-Verlag (3-540-00323-1), 2002, pp. 378–393.

[77] Wijsen, J. Project-Join-Repair: An Approach to Consistent Query Answering Under Functional Dependencies. In *Proc. 7th International Conference on Flexible Query Answering Systems (FQAS 06)*, 2006, pp. 1–12.