

UNIVERSITÀ DELLA CALABRIA



Dipartimento di ELETTRONICA,
INFORMATICA E SISTEMISTICA

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XXI ciclo

Tesi di Dottorato

Approximate Query Answering over Inconsistent Databases

Cristian Molinaro



UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XXI ciclo

Tesi di Dottorato

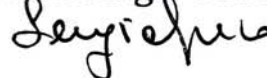
Approximate Query Answering over Inconsistent Databases

Cristian Molinaro
Cristian Molinaro

Coordinatore
Prof. Domenico Talia



Supervisore
Prof. Sergio Greco



DEIS

DEIS- DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
Novembre 2008

Settore Scientifico Disciplinare: ING-INF/05

to my mother

Preface

Integrity constraints have long been used to maintain database consistency, thus ensuring that every possible database reflects a valid, consistent state of the world. However, nowadays several applications have to deal with inconsistent databases, namely databases which violate given integrity constraints, because integrity constraints may not be enforced or satisfied. For instance, inconsistency may arise in *data integration*, where multiple autonomous sources are integrated together. Even if the sources are separately consistent, the integrated database may be inconsistent. Inconsistency may also occur when integrity constraints are unenforced because integrity checking is infeasible or too costly. There are plenty of other scenarios where inconsistency arises.

Dealing with inconsistent databases we face the problem of extracting reliable information from them. In this regard, most of the works in the literature are based on the *consistent query answering* (CQA) framework. This framework relies on the notions of *repair* and *consistent query answer*. Intuitively, a repair for a possibly inconsistent database is a consistent database which “minimally” differs from the original one. In general, there may be more than one repair for an inconsistent database. The *consistent answers* to a query over a possibly inconsistent database are those answers that can be obtained from every repair.

Several different dimensions of consistent query answering have been explored in the last years: different notions of repair minimality (leading to different semantics for consistent query answers); different classes of queries and integrity constraints; different methods of computing consistent query answers.

In this thesis, we address several issues regarding the problem of repairing and querying inconsistent databases; they are briefly introduced below.

In general, the consistency of an inconsistent database may be restored in different ways. In this scenario it is natural to express preferences among the updates which lead a database to a consistent state. For example, if a database violates a functional dependency because of conflicting data coming from different sources such conflicts may be resolved if the sources have different

reliability. Similarly, new information may be preferred to old information. In this regard, we propose *prioritized active integrity constraints*, a special type of active rules which allows us to express integrity constraints, feasible updates which should be performed (in order to restore consistency) when integrity constraints are violated and preferences among such updates.

Inconsistency leads to *uncertainty* as to the actual values of tuple attributes. Thus, it is natural to study the possible use of incomplete database frameworks in this context. The set of repairs for a possibly inconsistent database could be represented by means of an incomplete database whose possible worlds are exactly the repairs of the inconsistent database. In this thesis we address this issue considering a specific incomplete database framework: *disjunctive databases*.

We propose a framework for querying inconsistent databases in the presence of functional dependencies and foreign key constraints, where consistent answers for a particular class of conjunctive queries can be computed in polynomial time.

Computing consistent query answers is an intractable problem in the general case. In order to cope with this problem, we propose a framework, based on three-valued logic, which allows us to compute a sound and incomplete set of consistent query answers in polynomial time.

The original notion of repair has been criticized as *too coarse-grained*: deleting a tuple to remove an integrity violation potentially eliminates useful information in that tuple. Moreover, the original notion of consistent query answers does not allow us to discriminate among answers which are not consistent. In order to cope with the aforementioned problems, we propose a framework for querying inconsistent databases where both the notions of repair and query answer differ from the classical ones.

Main Contributions

The main contributions of the thesis are the following:

1. The problem of expressing preferences among repairs is addressed. Specifically, we propose a logical framework based on *prioritized active integrity constraints* (PAICs) for handling constraints with preferences. A PAIC allows us to express a universal integrity constraints, the feasible updates which should be performed whenever the constraint is violated and preferences among the feasible updates. Then, given a database and a set of PAICs, *founded* repairs are those repairs obtained by performing only feasible updates. Preferences among updates induce preferences among founded repairs, so that preferred repairs can be identified among founded repairs. The preferred query answers are those ones obtained from every preferred repair. We propose a technique for computing founded repairs which consists in deriving a disjunctive Datalog program from a set of

PAICs so that the stable models of the so obtained program correspond to founded repairs. We show that some desirable properties on the set of preferred repairs hold in the proposed framework.

2. We study the problem of *representing* the set of repairs of a possibly inconsistent database by means of a disjunctive database (i.e. a disjunctive database whose minimal models are the repairs).

We show that, given a database and a set of denial constraints, there exists a (unique) disjunctive database, called *canonical*, which represents the repairs of the database w.r.t. the constraints and is contained in any other disjunctive database with the same set of minimal models. We propose an algorithm for computing the canonical disjunctive database.

Moreover, we study the size of the canonical disjunctive database in the presence of functional dependencies for both set-repairs (consistent databases for which the symmetric difference from the original database is minimal under set inclusion) and card-repairs (consistent databases for which the cardinality of symmetric difference from the original database is minimal).

3. We investigate the problem of repairing and querying databases in the presence of (particular sets of) functional dependencies and foreign key constraints. We present a repairing strategy whereby only tuple updates and insertions are allowed in order to restore consistency: when foreign key constraints are violated, new tuples (possibly containing *null values*) are inserted into the database; when functional dependency violations occur, tuple updates (possibly introducing *unknown values*) are performed. We propose a semantics of constraint satisfaction for databases containing null and unknown values, since the repairing process can lead to such databases. The proposed approach allows us to obtain a unique repaired database which can be computed in polynomial time. The result of the repairing technique is an incomplete database (in particular, an OR-database). The semantics of query answering over an inconsistent database consists in computing *certain* answers on the repaired database. We also identify a class of conjunctive queries whose answers can be computed in polynomial time.
4. Since computing consistent query answers is in general an intractable problem, we propose a technique for computing a sound and incomplete set of consistent answers in polynomial time.

We propose *three-valued repairing strategy* relying on update operations which make the truth value of database atoms *true*, *false* or *undefined*. Thus, in this setting, three-valued databases are considered and a new semantics of constraint satisfaction (for three-valued databases) is proposed. For standard (two-valued) databases the proposed semantics coincides with the classical one. We show that the set of three-valued repairs defines a lower semi-lattice whose top elements are standard (two-valued) repairs and whose bottom element is called *deterministic repair*.

- We show that by evaluating a query over the deterministic repair we get sound, but not complete, consistent answers. We study some classes of queries and constraints for which the proposed technique is also complete. Moreover, we show that the deterministic repair and query answers can be computed in polynomial time, by showing that a logic program whose perfect model corresponds to the deterministic repair can be obtained by suitably rewriting the integrity constraints associated with the database.
5. Finally, we propose a framework for querying inconsistent databases which aims at preserving better the information in an inconsistent database and providing more informative query answers. In order to achieve these goals, we adopt notions of repair and query answer which differ from the classical ones. Specifically, the repairing strategy relies on value-updates, so that the information of an inconsistent database is preserved better than approaches based on tuple deletions. Answers to queries are tuples associated with probabilities: this approach allows us to provide more informative answers to queries over inconsistent databases. We also propose a technique for computing approximate probabilistic query answers in polynomial time.

Organization

The thesis is organized as follows. In Chapter 1 we introduce basic concepts and notations of first-order logic, disjunctive Datalog, relational databases and integrity constraints. In Chapter 2 we provide a survey of the literature on consistent query answering over inconsistent databases. In Chapter 3 we present a framework for expressing preferences in repairing inconsistent databases. In Chapter 4 we address the problem of representing a set of repairs by means of a disjunctive databases. In Chapter 5 we consider the problem of querying inconsistent databases in the presence of functional dependencies and foreign key constraints, and identify a class of conjunctive queries whose answers can be computed in polynomial time. Chapter 6 presents a framework, based on three-valued logic, for computing sound but not complete consistent query answers in polynomial time. In Chapter 7 we propose a framework for querying inconsistent databases where query answers are *probabilistic*. Finally, conclusions are drawn.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Prof. Sergio Greco, for his valuable guidance and support. He has been a great mentor and friend (and also great travelling companion), always present to guide me in the right direction.

I owe my gratitude to all my colleagues at DEIS department for their friendship and stimulating discussions. I also acknowledge Prof. Domenico Talia, for his care in coordinating the PhD course I attended, Giovanni Costabile and Francesco De Marte, for their care and attention.

I would like to thank Prof. Jan Chomicki for his warm hospitality during my staying in Buffalo.

Last, but definitely not least, I am very grateful to my parents, my sister and my brother for always standing by me.

Rende, November 2008

Cristian Molinaro

Contents

Preface	VII
Main Contributions	VIII
Organization	X
Acknowledgements	X
1 Preliminaries	1
1.1 First-Order Logic	1
1.2 Disjunctive Datalog	2
1.3 Relational Databases	4
1.4 Integrity Constraints	4
1.4.1 Universal Integrity Constraints	5
1.4.2 Denial Constraints	6
1.4.3 Functional Dependencies	6
1.4.4 Inclusion Dependencies	6
2 Consistent Query Answers over Inconsistent Databases	9
2.1 Introduction	9
2.2 Computing CQAs: Methods	11
2.2.1 Query Rewriting	12
2.2.2 Representing all Repairs	20
2.2.3 Logic Programs	22
2.3 Computing CQAs: Computational Complexity	31
2.4 Variants of CQA	32
2.5 Discussion	34
3 Prioritized Active Integrity Constraints	35
3.1 Introduction	35
3.2 Syntax and Semantics of PAICs	36
3.3 Computing Repairs Through Datalog Programs	42
3.4 Discussion	44

4	Disjunctive Databases for Representing Repairs	47
4.1	Introduction	47
4.2	Disjunctive Databases	49
4.3	Representing Repairs Through Disjunctive Databases	49
4.4	Functional Dependencies	53
4.5	Cardinality-based Repairs	60
4.6	Discussion	62
5	Polynomial Time Queries over Inconsistent Databases	63
5.1	Introduction	63
5.2	Repairing Inconsistent Databases	65
5.2.1	Semantics of Constraint Satisfaction	66
5.2.2	Repairing	68
5.3	Query Answering	72
5.4	Discussion	74
6	A Three-Valued Semantics for Querying and Repairing Inconsistent Databases	77
6.1	Introduction	77
6.2	Notation and Terminology	79
6.3	Partial Repairs	81
6.3.1	Constraint Satisfaction Under Partial Semantics	81
6.3.2	Repairing Databases	86
6.4	Query Answering	88
6.5	Computing the Deterministic Repair	93
6.6	Discussion	98
7	Approximate Probabilistic Query Answering over Inconsistent Databases	101
7.1	Introduction	101
7.2	Probabilistic Relational Model	103
7.3	Repairing	106
7.4	Query Answering	109
7.5	Discussion	113
	Conclusions	115
	References	119

Preliminaries

In this chapter, first-order logic, disjunctive Datalog, relational databases and integrity constraints are introduced. We assume that the reader is familiar with the aforementioned topics, so only basic concepts and notations used hereafter are presented (for more, see [1, 73]).

1.1 First-Order Logic

A first-order language L is defined over an alphabet Σ which consists of countable sets of *variable*, *predicate* and *function* symbols. A predicate (resp. a function) symbol is said to be a k -ary predicate (resp. function) symbol if the number of arguments that it takes is equal to k . We assume that the binary equality predicate symbol $=$ is defined. A 0-ary function symbol is called a *constant*. A first-order language is *function-free* if it contains only functions with arity equal to zero.

The family of *terms* over the alphabet Σ is recursively defined as follows: a constant or a variable is a term; $f(e_1, \dots, e_n)$ is a term if f is an n -ary function symbol and the e_i 's are terms.

The (*well-formed predicate calculus*) *formulas* over L are built using logical connectives (\neg, \wedge, \vee), quantifiers (\forall, \exists), terms and predicate symbols as follows: $p(e_1, \dots, e_n)$ is an *atomic formula* (or *atom*) if p is an n -ary predicate symbol and e_1, \dots, e_n are terms; atomic formulas also include expressions of the form $e_1 = e_2$ where e_1 and e_2 are terms; $\neg\varphi, \varphi \wedge \phi, \varphi \vee \phi, \exists x\varphi, \forall x\varphi$ are formulas if φ and ϕ are formulas and x is a variable.

Free and *bound* occurrences of variables in formulas are recursively defined in the following way: each variable occurrence in an atom is free; if ϕ is $\varphi_1 \vee \varphi_2$, then an occurrence of variable x in ϕ is free if it is free as an occurrence of φ_1 or φ_2 ; this is extended to the other connectives. If ϕ is $\exists x\varphi$ (resp. $\forall x\varphi$), then an occurrence of variable $y \neq x$ is free in ϕ if it is free in φ , whereas each occurrence of x is bound in ϕ . In addition, each occurrence of x in ϕ which is free in φ is said to be in the *scope* of $\exists x$ (resp. $\forall x$) at the beginning of ϕ .

A *sentence* is a well-formed formula that has no free variable occurrences. Sentences will also be called *closed* (first-order) formulas. A formula is *quantifier-free* if no quantifier occurs in it.

A term or a formula is *ground* if no variables occur in it. A *literal* is an atom A or a negated atom $\neg A$; in the former case it is positive, whereas in the latter negative. Two literals are *complementary*, if they are of the form A and $\neg A$, for some atom A .

An *interpretation* of a first-order language is a 4-tuple $I = \langle U, C, P, F \rangle$, where U is a nonempty set of abstract elements called the *universe (of discourse)* and C, P, F give meaning to the sets of constant, predicate and function symbols: C is a function from the constant symbols into U ; P maps each n -ary predicate symbol p into an n -ary relation over U , i.e. a subset of U^n ; F assigns to each k -ary function symbol f an actual function $U^k \rightarrow U$. An interpretation is *finite* if its universe of discourse is finite.

The *Herbrand Universe* of a first-order language L is the set of all ground terms that can be constructed using constant and function symbols of L (if the language has no constants, then it is extended by adding an arbitrary new constant). The *Herbrand Base* of L is the set of all ground atoms constructed from the predicates of L and the ground terms from the Herbrand Universe as arguments. The Herbrand Universe and the Herbrand Base are both enumerable, and infinite if there is a function symbol of arity greater than zero. An *Herbrand interpretation* is a subset of the Herbrand Base. It is an interpretation where the universe of discourse is the the Herbrand Universe, all terms are interpreted as themselves, and each predicate symbol is mapped into a subset of the Herbrand Base.

The notion of *satisfaction* of a formula by an interpretation is defined in the standard way. An interpretation I is a *model* of a set Φ of sentences, denoted as $I \models \Phi$, if I satisfies each sentence in Φ . For a sentence ψ (resp. a set Φ of sentences), an *Herbrand model* is an Herbrand interpretation satisfying ψ (resp. every sentence in Φ). If Φ has no models, then Φ is said to be *inconsistent* or *unsatisfiable*; otherwise it is said to be *consistent* or *satisfiable*.

We say that a sentence ψ (logically) *implies* (or *supports*) ν , denoted as $\psi \models \nu$, if every model of ψ is a model of ν too, and ψ is (logically) *equivalent* to ν , denoted as $\psi \equiv \nu$, if the set of models of ψ is equal to the set of models of ν . A (first-order) *theory* is a set of sentence of a (first-order) language.

1.2 Disjunctive Datalog

Datalog is a well-known database query language. Basically, it is a function-free first-order language. Next, we briefly introduce *Disjunctive Datalog*, which extends Datalog with disjunction and negation. A (*disjunctive Datalog*) *rule* r is of the form¹:

¹ The meaning of the symbols ‘ \wedge ’ and ‘ $,$ ’ is the same

$$\bigvee_{i=1}^p A_i \leftarrow \bigwedge_{j=1}^m B_j, \bigwedge_{k=m+1}^n \text{not } D_k, \varphi \quad (1.1)$$

where $p > 0, n \geq 0$, the A_i 's, the B_j 's and the D_k 's are atoms, and φ is a conjunction of built-in atoms of the form $u\theta v$ where u and v are terms and θ is a comparison predicate. The disjunction on the left-hand side of the rule is called the *head* of r (denoted by $Head(r)$), whereas the conjunction on the right-hand side is called the *body* of r (denoted by $Body(r)$). A rule is *safe* if every variable occurring in it is safe; a variable x is safe if occurs in a positive literal of the body or in a built-in atom of the form $x = y$ where y is either a constant or a safe variable. The expression $H \leftarrow (B_{1,1} \vee \dots \vee B_{1,m_1}), \dots, (B_{n,1} \vee \dots \vee B_{n,m_n})$ can be used as shorthand for the rules $H \leftarrow B_{1,i_1}, \dots, B_{n,i_n}$ where $1 \leq i_j \leq m_j$ for $j = 1..n$.

A (*disjunctive Datalog*) *program* is a finite set of rules. A *not*-free (resp. *or*-free) program is called *positive* (resp. *normal*).

The Herbrand Universe (resp. Herbrand Base) of a program P , denoted by U_P (resp. B_P), is the set of all constants appearing in P (resp. the set of all ground atoms constructed from the predicate symbols appearing in P and the constants from U_P).

A rule or a program is *ground* if no variables occur in it. A rule r' is a *ground instance* of a rule r if r' is obtained from r by replacing every variable in r with some constant in U_P ; $ground(P)$ denotes the set of all ground instances of the rules in P .

An interpretation I for a program P is any subset of B_P ; I is a model of P if it satisfies all rules in $ground(P)$. The (model-theoretic) semantics of positive programs assigns to P the set of its *minimal models*, denoted by $\mathcal{MM}(P)$, where a model M for P is minimal if no proper subset of M is a model for P . For any interpretation I , P^I is the ground positive program derived from $ground(P)$ by 1) removing all rules that contain a negative literal $\text{not } A$ in the body s.t. $A \in I$, and 2) removing all negative literals from the remaining rules. An interpretation I is a (*disjunctive*) *stable model* of P if and only if $I \in \mathcal{MM}(P^I)$. The stable model semantics assigns to P the set $\mathcal{SM}(P)$ of its stable models. It is well known that stable models are minimal models (i.e. $\mathcal{SM}(P) \subseteq \mathcal{MM}(P)$) and that for positive programs minimal and stable model semantics coincide (i.e. $\mathcal{SM}(P) = \mathcal{MM}(P)$).

We now introduce a class of programs where the use of negation is restricted. A program P is called *stratified* if it is possible to partition the (finite) set S of all predicate symbols in P into sets $\{S_1, \dots, S_r\}$, called *strata*, such that for every rule of the form (1.1) in P there exists a constant c , $1 \leq c \leq r$, such that for every A_i $Stratum(A_i) = c$, for every B_j $Stratum(B_j) \leq c$ and for every D_k $Stratum(D_k) < c$, where $Stratum(A) = i$ iff the predicate symbol of the atom A is in the stratum S_i . Any partition $\{S_1, \dots, S_r\}$ of S satisfying the previous conditions is called a *stratification* of P . We use the term *stratification of the rules of P* to refer to the partition $\{P_1, \dots, P_r\}$ of P s.t. P_i contains the rules defining the predicates in S_i (a rule defines a

predicate p if an atom whose predicate symbol is p occurs in the head of the rule). A stratified normal program admits a unique stable model.

In the following, rules of the form

$$\leftarrow L_1, \dots, L_n$$

where the L_i 's are literals, will be used as a shorthand notation for

$$p \leftarrow L_1, \dots, L_n, \text{not } p$$

where p is a propositional symbol not appearing elsewhere. The previous rule emulates the integrity constraint $\text{false} \leftarrow L_1, \dots, L_n$ as the conjunction L_1, \dots, L_n must be false in every stable model.

1.3 Relational Databases

We assume that there exist countably infinite sets **att**, **relnam** and **dom** (called the underlying *domain*) of *attributes*, *relation names* and *constants*, respectively. The previous sets are pairwise disjoint. A *relation schema* is of the form $p(A_1, \dots, A_n)$ where $p \in \mathbf{relnam}$ and every $A_i \in \mathbf{att}$ (a schema can be also of the form $p(U)$, where U is a set of attributes). A (relational) *database schema* is a nonempty set of relation schemata. When different attributes should have distinct domains, we assume a mapping Dom on **att**, where $Dom(A)$ is a set of constants called the domain of the attribute A .

A *tuple* over a relation schema $p(A_1, \dots, A_n)$ is a mapping assigning to each attribute A_i an element in $dom(A_i)$, i.e. it is a list of values $\langle a_1, \dots, a_n \rangle$ where every a_i is the value of the tuple on the attribute A_i . The value of a tuple t on an attribute A_i will be denoted as $t[A_i]$. For a set of attributes $W = \{A_i, \dots, A_j\}$, $t[A_i, \dots, A_j]$ (also denoted as $t[W]$) is $\langle t[A_i], \dots, t[A_j] \rangle$.

A *relation instance* (or simply *relation*) over a relation schema $p(U)$ is a set of tuples over $p(U)$. In the following, a tuple $t = \langle a_1, \dots, a_n \rangle$ over $p(U)$, will also be denoted by $p(a_1, \dots, a_n)$ (or $p(t)$) since under a logic-programming perspective it is a fact (ground atom) over p . A *database instance* (or simply *database*) over a database schema $\{p_1(U_1), \dots, p_m(U_m)\}$ is a set of relations $\{r_1, \dots, r_m\}$ where each r_i is a relation over the schema $p_i(U_i)$. Since each tuple t in a relation p can be viewed as a fact $p(t)$, a database can be viewed as a finite set of facts.

1.4 Integrity Constraints

Integrity constraints express semantics information over data, i.e. properties, relationships that are supposed to be satisfied among data and they are mainly used to validate database transactions. They are usually defined by first-order

formulas or by means of special notations for particular classes such as functional and inclusion dependencies.

An *integrity constraint* (or *dependency*) is a first-order sentence of the form:

$$(\forall X)[\Phi(X) \rightarrow (\exists Z)\Psi(Y)]$$

where X , Y and Z are sets of variables, $Z = Y - X$, Φ is a (possibly empty) conjunction of atoms and Ψ is a nonempty conjunction of atoms. Equality atoms of the form $x_i = x_j$, where x_i and x_j are variables, can occur in an integrity constraint. Without loss of generality, we assume that no equality atom occurs in Φ and no existentially quantified variable participates in an equality atom in Ψ . Common classifications of dependencies are as follows:

- *Full* versus *embedded*: A full dependency is a dependency that has no existential quantifiers.
- *Single head* versus *multi-head*: A dependency is single head if the right hand formula involves a single atom and is multi-head otherwise.
- *Tuple generating* versus *equality generating*: A tuple generating dependency is a dependency in which no equality atoms occur; an equality generating dependency is a dependency for which the right-hand formula is a single equality atom (observe that an equality generating dependency is single head and full).
- *Typed* versus *untyped*: A dependency is typed if there is an assignment of variables to column positions such that (1) variables in relation atoms occur only in their assigned position, and (2) each equality atom involves a pair of variables assigned to the same position.
- *Unirelational* versus *Multirelational*: A dependency is unirelational if at most one relation name is used and is multirelational otherwise.

Most of the dependencies developed in database theory are restricted cases of some of the above classes. For instance, functional dependencies are unirelational, equality-generating dependencies. In the following, if not differently stated, we will assume that a given set of constraints is *satisfiable* (or *consistent*), that is there is a database instance that makes it true.

Given a database schema DS and a set IC of integrity constraints on DS , an instance D of DS is said to be *consistent* w.r.t. IC if $D \models IC$ in the standard model-theoretic sense, *inconsistent* otherwise ($D \not\models IC$).

Next, we present some particular classes of constraints which will be of interest in the following.

1.4.1 Universal Integrity Constraints

Universal integrity constraints are sentences of the form

$$\forall X[A_1 \vee \dots \vee A_m \vee \neg B_1 \vee \dots \vee \neg B_n \vee \varphi]$$

where the A_i 's and the B_j 's are atoms, φ is a conjunction of built-in atoms, X denotes the list of all variables appearing in the B_j 's; variables appearing in the A_i 's and in φ also appear in the B_j 's. Universal integrity constraints written under the previous form are said to be in *standard format*, even though they can be rewritten as follows

$$\forall X[B_1 \wedge \dots \wedge B_n \wedge \phi \rightarrow A_1 \vee \dots \vee A_m]$$

where ϕ is the negation of φ ; in turn, the previous constraint can be rewritten by moving literals from the head to the body and vice versa. Universal constraints with at most two database literals (i.e., literals whose predicate symbols are relation names) will be called *binary universal constraints*.

1.4.2 Denial Constraints

Denial constraints are universal integrity constraints of the form

$$\forall X[\neg B_1 \vee \dots \vee \neg B_n \vee \varphi]$$

that is only negative literals appear in the standard format of the universal constraint. Denial constraints with $n \leq 2$ will be called *binary denial constraints*.

1.4.3 Functional Dependencies

Functional dependencies are a special case of binary denial constraints. They are typed, unirelational, equality-generating constraints of the form

$$\forall X_1, X_2, X_3, X_4, X_5[p(X_1, X_2, X_4) \wedge p(X_1, X_3, X_5) \rightarrow X_2 = X_3]$$

where the X_i 's are tuple of variables. A more familiar formulation of the above functional dependencies is $p[V] \rightarrow p[W]$ (or simply $V \rightarrow W$ if the relation symbol is understood from the context), where V is the set of attributes of p corresponding to X_1 and W is the set of attributes of p corresponding to X_2 (and X_3). Given a relation schema $p(U)$ and a set FD of functional dependencies over it, a *key* of p is a minimal (under \subseteq) set $K \subseteq U$ of attributes such that FD entails $K \rightarrow U$. In this case, we say that each $K \rightarrow W$ in FD is a key dependency. If, additionally, K is the primary (one designed) key of p , then $K \rightarrow W$ is called *primary key dependency*.

1.4.4 Inclusion Dependencies

An *inclusion dependency*, also known as *referential integrity constraint*, is a sentence of the form

$$\forall X \exists Z[p(X) \rightarrow q(Y, Z)]$$

where X and Z are disjoint sets of variables and $Y \subseteq X$. The foregoing inclusion dependency can be written also as $p[V] \subseteq q[W]$, where V (resp. W) is the set of attributes of p (resp. q) corresponding to Y . Full inclusion dependencies are those expressible without the existential quantifier (they are a special case of binary universal constraints). For instance, $\forall X, Y [p(X, Y) \rightarrow q(X)]$ is full. Given an inclusion dependency $p[V] \subseteq q[W]$, if W is a key of q then the dependency is a *foreign key constraint*. If, additionally, W is the primary key of q , then the dependency is called *primary foreign key constraint*. Let ID be a set of inclusion dependencies over a database schema DS . Consider a directed graph whose vertices are relation names from DS and such that there is an edge (p, q) in the graph iff there is an inclusion dependency $p[V] \subseteq q[W]$ in ID . ID is *acyclic* if the above graph does not have a cycle.

Consistent Query Answers over Inconsistent Databases

The problem of extracting reliable information from inconsistent databases has been extensively studied in the past several years. Most of the works in the literature rely on the *consistent query answering* (CQA) framework [4]. In this chapter we survey the literature on consistent query answering over inconsistent databases. Specifically, we first introduce the basic notions of *repair* and *consistent query answer*. Next, we present the main techniques for computing consistent answers to queries. Then we present complexity results of consistent query answering for different classes of queries and constraints. Finally, we briefly summarize works in the literature where the notions of repair and consistent query answer differ from the original ones.

2.1 Introduction

Integrity constraints have long been used to maintain database consistency, thus ensuring that every possible database reflects a valid, consistent state of the world. However, nowadays inconsistent databases arise in several scenarios because integrity constraints may not be enforced or satisfied. For example, when multiple autonomous data sources are integrated together, even if the sources are separately consistent, the integrated database may be inconsistent. In some contexts, integrity constraints may be unenforced because integrity checking is too costly or infeasible. Dealing with inconsistent databases, we face the problem of extracting reliable information from them. Different approaches have been developed in order to deal with inconsistency in a flexible manner, they are illustrated in the following example.

Example 2.1. Consider a database schema consisting of two unary relations p_1 and p_2 , and the denial constraint $\forall x(\neg p_1(x) \vee \neg p_2(x))$. Consider a database consisting of the following facts: $p_1(a), p_1(b), p_2(a)$. Under *prevention* (usual constraint enforcement), such instance could not arise: only one of $p_1(a)$ and

$p_2(a)$ could be inserted into the database. Under *ignorance* (constraint non-enforcement), no distinction is made between $p_1(a)$ and $p_1(b)$, despite that the latter, not being involved in a constraint violation, appears to represent more reliable information. Under *isolation* [18], both $p_1(a)$ and $p_2(a)$ are dropped (or ignored in query answering). Under *weakening* [9, 62], $p_1(a)$ and $p_2(a)$ are replaced by $p_1(a) \vee p_2(a)$. Allowing *exceptions* [15] means that the constraint is weakened to $\forall x(\neg p_1(x) \vee \neg p_2(x) \vee x = a)$, but query answering is not affected.

Another approach relies on the notions of *repair* and *consistent query answer*, firstly proposed in [4]. A repair of a database w.r.t. a set of integrity constraints is a consistent database which *minimally* differs from the original one; the distance between two databases D_1 and D_2 is expressed by means of the symmetric difference $\Delta(D_1, D_2) = (D_1 - D_2) \cup (D_2 - D_1)$. Consistent answers to a query over a possibly inconsistent database are those answers which can be derived from every repair of the original database. We show the basic ideas underlying consistent query answering in the following example.

Example 2.2. Let $employee(Name, Salary, Dept)$ be a relation schema with the functional dependency $fd : Name \rightarrow Salary Dept$ stating that each employee has a unique salary and a unique department. Consider the following relation instance r :

<i>Name</i>	<i>Salary</i>	<i>Dept</i>
<i>john</i>	50	<i>cs</i>
<i>john</i>	100	<i>cs</i>

Clearly, r is inconsistent w.r.t. f as it stores two different salaries for the same employee $john$. There exist two repairs for r and f , namely $\{employee(john, 50, cs)\}$ (obtained by deleting the second tuple in r) and $\{employee(john, 100, cs)\}$ (obtained by deleting the first tuple in r). The consistent answer to the query asking for the department of $john$ is cs (as this is the answer of the query in both the repairs) whereas the query asking for the salary of $john$ has no consistent answer (as the two repairs do not agree on the answer).

Several different dimensions of consistent query answering have been explored after [4]:

- different notions of repair minimality (leading to different semantics for consistent query answers);
- different classes of queries and integrity constraints;
- different methods of computing consistent query answers.

In this chapter, we will survey techniques for computing consistent query answers, main complexity results on this topic and variants of the original framework (an introduction to the central concepts of consistent query answering is [24], whereas surveys on this topic are [13, 11]).

We now present the formal definitions of repair and consistent query answer introduced in [4].

Definition 2.1. *Repair.* Given a set IC of integrity constraints and two databases D and R , we say that R is a *repair* of D w.r.t. IC if $R \models IC$ and there is no database D' such that $D' \models IC$ and $\Delta(D', D) \subset \Delta(R, D)$.

We denote by $rep(D, IC)$ the set of repairs of D w.r.t. IC . This set is nonempty for satisfiable sets of constraints.

Definition 2.2. *Consistent Query Answer (CQA).* Let q , D and IC be respectively a query, a database and a set of integrity constraints. A tuple t is a *consistent answer* to q in D w.r.t. IC iff t is an answer to q in every repair of D w.r.t. IC . We denote by $consistent_{IC}(q, D)$ the set of consistent answers to q on D w.r.t. IC . Likewise, given a boolean query q' , we define *true* being a consistent answer to q' on D w.r.t. IC , denoted by $D \models_{IC} q'$, if $R \models q'$ for every repair R of D w.r.t. IC .

Two basic decision problems are:

- *repair checking:* Is a database a repair of another one w.r.t. the integrity constraints?
- *consistent query answering:* Is a tuple a consistent query answer to a query on a database w.r.t. the integrity constraints?

In this dissertation, we will consider only the latter problem. The *data complexity* assumption is adopted [1, 75], that is the complexity of the problem is measured as a function of the number of tuples in a database; the query and the integrity constraints are considered fixed.

2.2 Computing CQAs: Methods

We note that already in the presence of a single primary key dependency there are inconsistent relations with exponentially many repairs, as shown in the following example.

Example 2.3. Consider the relation schema $r(A, B)$ with the functional dependency $A \rightarrow B$. Let D_n ($n > 0$) be the family of databases, containing $2n$ tuples, of the following form:

A	B
a_1	b_1
a_1	b_2
\vdots	\vdots
a_n	b_1
a_n	b_2

It is easy to see that each database in D_n admits 2^n repairs.

Thus, computing consistent query answers by applying directly Definition 2.2 is impractical. Nevertheless, several practical mechanisms for the computation of consistent query answers without computing all repairs have been developed: query rewriting [4, 41, 42, 39, 40], compact representations of repairs [26, 27], and logic programs [7, 20, 49, 50]. The first is based on rewriting the input query q into a query q_{IC} such that the evaluation of q_{IC} returns the set of consistent answers to q . This method works only for restricted classes of queries and constraints. The second method relies on a compact representation of the integrity constraint violations which is used during query evaluation. The third approach uses disjunctive logic programs to specify all repairs, and then with the help of a disjunctive logic programming system (e.g. [59]) finds the consistent answers to a given query. Although this approach is applicable to very general queries in the presence of universal constraints, the complexity of evaluating disjunctive logic programs makes this method impractical for large databases.

In this section we present the main proposed techniques for computing consistent query answers, according to the aforementioned classification.

We observe that the techniques proposed in this thesis for computing query answers over inconsistent databases (see Chapter 6 and Chapter 7) adopt the approach of computing “approximate” query answers, rather than “exact” query answers, in polynomial time. In Chapter 6 we will present a polynomial time technique for computing sound but incomplete consistent query answers. In Chapter 7 we will present an approach for computing *approximate probabilistic query answers* over inconsistent databases in polynomial time (here the semantics of query answering differs from the original one).

2.2.1 Query Rewriting

The query rewriting approach for computing consistent query answers consists in rewriting a query q w.r.t. a set IC of integrity constraints into a query, say it q_{IC} , s.t. for every database D the consistent answers to q on D w.r.t. IC are obtained by evaluating q_{IC} directly on D (thus, without computing the repairs).

First-Order Query Rewriting

In [41, 42] an algorithm for computing consistent answers has been proposed. Specifically, given a query q , the algorithm returns a first-order query Q such that for every possibly inconsistent database D , the consistent answers to q on D can be obtained by evaluating Q directly over D . The algorithm works for a particular class of conjunctive queries (which will be precisely characterized below) and constraints which consists of one key dependency per relation; the algorithm runs in linear time in the size of the query. Moreover, the same papers show a class of queries s.t. for every query in the class, either the consistent query answering problem is in PTIME or it is coNP-complete.

Specifically, for this class of queries, the conditions of applicability of the algorithm (which can be verified in polynomial time in the size of the query) are necessary and sufficient in order to the consistent query answer problem be in PTIME.

In the rest of this section, only conjunctive queries without repeated relation symbols are considered, that is every relation symbol occurs in a query at most once. It is worth noting that, the problem of consistent query answering for conjunctive queries in the presence of one key constraint per relation is known to be coNP-complete in general [19, 25]. This is the case even for queries with no repeated relation symbols. Next, a class of conjunctive queries for which the problem of computing consistent answers is tractable is characterized. Let us introduce some definitions which are used to define the aforementioned class of conjunctive queries.

Let q be a conjunctive query. The *join graph* G of q is a directed graph such that (i) the vertices of G are the literals of q , (ii) there is an arc from r_i to r_j if $i \neq j$ and there is some existentially-quantified variable w in q s.t. w occurs at the position of a nonkey attribute in r_i and occurs in r_j too (in any position).

We say that there is a *join* on a variable w of q if w appears in two literals $r_i(X_i, Y_i)$ and $r_j(X_j, Y_j)$ s.t. $i \neq j$ (X_i and X_j are those variables corresponding to the key attributes of r_i and r_j , respectively). If w occurs in Y_i and Y_j , we say that there is a *nonkey-to-nonkey* join on w ; if w occurs in Y_i and X_j , we say that there is a *nonkey-to-key* join; if w occurs in X_i and X_j , we say that there is a *key-to-key* join. Given two literals $r_i(X_i, Y_i)$ and $r_j(X_j, Y_j)$ of a query, we say that there is a *full nonkey-to-key* join from r_i to r_j if every variable of X_j appears in Y_i .

We now define the class of (tractable) conjunctive queries for which the algorithm produces a rewriting.

Definition 2.3. *\mathcal{C}_{forest} query.* Let q be a conjunctive query without repeated relation symbols and all of whose nonkey-to-key joins are full. Let G be the join graph of q . We say that $q \in \mathcal{C}_{forest}$ if G is a forest (i.e., every connected component of G is a tree).

The rewriting algorithm is correct for queries in \mathcal{C}_{forest} : let q be a query belonging to \mathcal{C}_{forest} , D be a database over a schema DS with a set IC of integrity constraints consisting of one key dependency per relation, Q be the first-order query returned by the algorithm; then, a tuple t is an answer to Q over D iff $t \in consistent_{IC}(q, D)$. We next present some examples to show the basic ideas underlying the algorithm (for more details, see [42]).

Example 2.4. Consider the database schema $\{r_1(A, B)\}$ where A is the primary key of r_1 . Let $q = \exists x.r_1(x, a)$, where a is a constant. Consider the (inconsistent) database $D_1 = \{r_1(c_1, a), r_1(c_1, b)\}$. It is easy to see that $D_1 \models q$. However, $consistent_{IC}(q, D_1)$ (clearly, IC refers to the key constraint) is *false* since the repair $\{r_1(c, b)\} \not\models q$. Now, consider $D_2 = \{r_1(c_1, a), r_1(c_1, b), r_1(c_2, a)\}$.

It is easy to see that $\text{consistent}_{IC}(q, D_2)$ is *true*. This is because there is a key value in r_1 (c_2 in this case) that appears with a as its nonkey value, and does not appear with any other constant a' such that $a' \neq a$. This can be checked with a formula $Q_{\text{consistent}}(x) = \forall y'. r_1(x, y') \rightarrow y' = a$. Indeed, the query rewriting Q for q is the conjunction of q and $Q_{\text{consistent}}$:

$$Q = \exists x. r_1(x, a) \wedge \forall y'. r_1(x, y') \rightarrow y' = a$$

Example 2.5. Consider the database schema $\{r_1(A, B), r_2(C, D)\}$ where A is the primary key of r_1 and C is the primary key of r_2 . Let $q = \exists x, y, z. r_1(x, y) \wedge r_2(y, z)$. Consider the (inconsistent) database $D_1 = \{r_1(c_1, d_1), r_1(c_1, d_2), r_2(d_1, e_1)\}$. It is easy to see that $D_1 \models q$. However, $\text{consistent}_{IC}(q, D_1)$ is *false*. Now, consider $D_2 = \{r_1(c_1, d_1), r_1(c_1, d_2), r_2(d_1, e_1), r_2(d_2, e_2)\}$. It is easy to see that $\text{consistent}_{IC}(q, D_2)$ is *true*, since every nonkey value that appears together with c_1 in some tuple of r_1 (in this case, d_1 and d_2) joins with a tuple of r_2 . This can be checked with a formula $Q_{\text{consistent}}(x) = \forall y. r_1(x, y) \rightarrow \exists z. r_2(y, z)$. The query rewriting algorithm yields the following query (which is the conjunction of q and $Q_{\text{consistent}}$):

$$\exists x, y, z. r_1(x, y) \wedge r_2(y, z) \wedge \forall y. (r_1(x, y) \rightarrow \exists z. r_2(y, z))$$

In general, the algorithm returns a first-order query Q that is obtained as the conjunction of the input query q and a new query called $Q_{\text{consistent}}$. The query $Q_{\text{consistent}}$ is used to ensure that q is satisfied in every repair. It is important to notice that $Q_{\text{consistent}}$ will be applied directly to the original (possibly inconsistent) database, that is repairs will not be generated.

In [41, 42], it has been shown that minimal relaxations of the conditions characterizing the class $\mathcal{C}_{\text{forest}}$ lead to intractability. In particular, it has been shown the intractability of the consistent query answering problem for (1) a conjunctive query whose join graph has a cycle of length two, and (2) a conjunctive query whose join graph is a forest, but the query has some nonkey-to-key joins that are not full. The former query is

$$\exists x, x', y. s_1(x, y) \wedge s_2(x', y)$$

and the database schema is $s_1(A, B), s_2(C, D)$ where A and C are the primary keys of s_1 and s_2 , respectively. The latter query is

$$\exists x, x', w, w', z, z', m. r_1(x, w) \wedge r_2(m, w, z) \wedge r_3(x', w') \wedge r_4(m, w', z')$$

where the database schema is $r_1(A, B), r_2(C, D, E), r_3(F, G), r_4(H, I, L)$ with A, CD, F, HI primary keys of r_1, r_2, r_3, r_4 , respectively.

We now present the dichotomy result. Let q be a conjunctive query without repeated relation symbols and all of whose nonkey-to-key joins are full. We say that q is in the class \mathcal{C}^* if for every pair A and A' of literals of q at most one of the following conditions holds:

- there is a key-to-key join between A and A' ,

- there is a nonkey-to-nonkey join between A and A' ,
- there are literals A_1, \dots, A_m in q such that there is a nonkey-to-key join from A to A_1 , from A_m to A' , and from A_i to A_{i+1} , for every i such that $1 \leq i < m$.

We say that a query q is in class \mathcal{C}_{hard} if $q \in \mathcal{C}^*$ and $q \notin \mathcal{C}_{forest}$. The consistent query answering problem for every query in \mathcal{C}_{hard} is coNP-complete; then for every query in \mathcal{C}^* the CQA problem is either PTIME or coNP-complete.

Rewriting for SQL Queries

The papers [39, 40] present the *ConQuer* system which computes consistent answers to SQL queries. Specifically, a special class of select-project-join (SPJ) SQL queries with aggregation, grouping, set or bag semantics is considered (this class will be precisely defined below), whereas the constraints consist of at most one key constraint per relation. For set semantics, we are only concerned with finding the set of tuples that occur in the query result for every repair; under bag semantics, the multiplicity of a tuple in the consistent query answer is the minimum multiplicity from any repair. The system relies on a technique which rewrites SQL queries into SQL queries in such a way that the latter allows us to retrieve the consistent answers to the original queries.

First, we will consider SPJ queries without aggregation or grouping. To define the class of queries that can be handled by ConQuer, we first introduce the notion of *join graph*. Given a SQL query q , the *join graph* G of q is a directed graph such that:

- the vertices of G are the relations used in q ,
- there is an arc from r_i to r_j if a non-key attribute of r_i is equated with a key attribute of r_j .

Definition 2.4. *Tree query.* We say that a select-project-join-group-by query q is a tree query if (1) every join condition of q involves the key of at least one relation; (2) the join graph of q is a tree. We consider only queries containing equi-joins (no inequality joins). The selection conditions may contain comparisons (e.g., $<$) or functions. Each relation may be used at most once in the query. The query may contain aggregate expressions.

Note that the previous definition restricts the nonkey-to-key joins of the query to be acyclic, and does not permit non-key to non-key joins (since every join must involve the key of a relation).

The rewriting technique consists in generating a subquery that retrieves “candidates” consistent answers and another subquery which allows us to filter out from the result of the former the answers which are not actually consistent. The former subquery is a slight modification of the original one: the observation here is that with key constraints and monotone queries, the

evaluation of the original query on the source database gives a set of possible answers (which is a superset of the consistent answers). The next example illustrates the rewriting approach.

Example 2.6. Consider the following database D :

<i>order</i>			<i>customer</i>	
<i>orderkey</i>	<i>clerk</i>	<i>custfk</i>	<i>custkey</i>	<i>acctbal</i>
o1	ali	c1	c1	2000
o2	jo	c2	c1	100
o2	ali	c3	c2	2500
o3	ali	c4	c3	2200
o3	pat	c2	c3	2500
o4	ali	c2		
o4	ali	c3		
o5	ali	c2		

over the schema $\{order(orderkey, clerk, custfk), customer(custkey, acctbal)\}$ where *orderkey* and *custkey* are primary keys. Consider the following query q which retrieves the clerks who have processed orders for customers with a balance over 1000:

```
SELECT o.clerk
FROM customer c, order o
WHERE c.acctbal>1000 AND o.custfk=c.custkey
```

A query rewriting Q that computes the consistent answers to q is the following:

```
SELECT clerk
FROM Candidates Cand
WHERE NOT EXISTS ( SELECT * from Filter F
                   WHERE Cand.orderkey = F.orderkey)
```

where *Candidates* and *Filter* are the following subqueries:

```
Candidates AS ( SELECT DISTINCT o.orderkey, o.clerk
                FROM customer c, order o
                WHERE c.acctbal > 1000 and o.custfk=c.custkey)
```

```
Filter AS ( SELECT o.orderkey
            FROM Candidates Cand
              JOIN order o ON Cand.orderkey = o.orderkey
              LEFT OUTER JOIN customer c ON o.custfk=c.custkey
            WHERE c.custkey IS NULL OR c.acctbal ≤ 1000 )
UNION ALL
SELECT orderkey
FROM Candidates Cand
GROUP BY orderkey
HAVING COUNT(*) > 1)
```

The **Candidates** subquery corresponds to the original query except that it uses the **DISTINCT** keyword and the attribute *orderkey* of the relation *order* has been introduced into the **SELECT** clause. In general, the **Candidates** subquery is obtained from the original one by introducing the **DISTINCT** keyword and the key attributes of the relation at the root of the join graph of the query into the **SELECT** clause. The evaluation of **Candidates** on *D* gives the following answers $\{(o1, ali), (o2, jo), (o2, ali), (o3, pat), (o4, ali), (o5, ali)\}$.

The subquery **Filter** returns the orders that should be filtered out from the result of **Candidates** because they are not consistent answers. In this case, **Filter** returns the orders $\{o1, o2, o3\}$. The orders *o1* and *o3* are filtered out by the former subquery of **Filter**, whereas the order *o2* by the latter. The order *o1* is returned by **Filter** because the customer of the order, namely *c1*, has not an account balance greater than 1000 for sure. The order *o3* is returned by **Filter** because it appears in a tuple (the fourth of *order*) which does not join with any tuple of *customer*. Observe that **Filter** computes a left-outer join between *order* and *customer*. Therefore *o3* appears together with a null value for attribute *custkey* in the left-outer join. The order *o2* is filtered out because its clerk maybe either *jo* or *ali* and thus *o2* should not contribute with its clerks to the consistent query answer of *q*. Observe that, detecting of the cases in which non-key to key joins are not satisfied (as for order *o3*) is obtained performing a left-outer join rather than an inner join. This can be done because we are considering queries whose join graph is a tree. Specifically, the left-outer join of the relations is obtained starting at the relation at the root of the join graph (tree), and recursively traversing it in the direction of its edges, that is, from a relation joined on a non-key attribute to a relation joined on its key.

Q takes the tuples of the result of **Candidates** whose order is not retrieved by **Filter**, and projects on the clerk attribute. The final result is $\{ali, ali\}$, which is the consistent answer. Notice that the *Q* computes not only the fact that *ali* is a consistent answer, but also the correct multiplicity.

We now present the rewriting method for tree queries that may have grouping and aggregation. We will consider SQL queries of the following form:

```
SELECT G, agg1(e1) AS E1, ..., aggn(en) AS En
FROM F
WHERE W
GROUP BY G
```

where *G* is the set of attributes we are grouping on, and *agg*₁(*e*₁), ..., *agg*_{*n*}(*e*_{*n*}) are aggregate expressions with functions *agg*₁, ..., *agg*_{*n*}, respectively (a function may be **MAX**, **MIN**, **SUM**). We will assume that the select clause renames the aggregate expressions to *E*₁, ..., *E*_{*n*}. Notice that we are focusing on queries where all the attributes in the group by clause appear in the select clause. This is a restriction because, in general, SQL queries may have some attributes in

the group by clause which do not appear in the select clause (although not vice versa).

The semantics of query answering is that one proposed in [5], namely a range for each value of G that is a consistent answer is given (observe that [5] considers queries with just one aggregated attribute and no grouping). Before formally presenting the semantics of query answering, we define the query q_G as the query obtained from q by removing all the aggregate expressions from the SELECT clause, that is q_G is of the form:

```
SELECT  $G$ 
FROM  $\mathcal{F}$ 
WHERE  $\mathcal{W}$ 
GROUP BY  $G$ 
```

Definition 2.5. *Range-Consistent Query Answer.* Let D be a database, q be a query, and IC be a set of integrity constraints. We say that (\mathbf{t}, \mathbf{r}) is a *range-consistent query answer* to q on D if

1. \mathbf{t} is a consistent answer for q_G on D ; and
2. $\mathbf{r} = (\min_{E_1}, \max_{E_1}, \dots, \min_{E_n}, \max_{E_n})$, and for each i such that $1 \leq i \leq n$:
 - $\min_{E_i} \leq \pi_{E_i}(\sigma_{G=\mathbf{t}}(q(R))) \leq \max_{E_i}$, for every repair R ; and
 - $\pi_{E_i}(\sigma_{G=\mathbf{t}}(q(R))) = \min_{E_i}$, for some repair R ; and
 - $\pi_{E_i}(\sigma_{G=\mathbf{t}}(q(R))) = \max_{E_i}$, for some repair R .

Example 2.7. Consider the following database D :

<i>customer</i>			
<i>custkey</i>	<i>nationkey</i>	<i>mktsegment</i>	<i>acctbal</i>
c1	n1	building	1000
c1	n1	building	2000
c2	n1	building	500
c2	n1	banking	600
c3	n2	banking	100

where *custkey* is the primary key. Consider the following query q , which retrieves the total account balance for customers in the building sector, grouped by nation:

```
SELECT nationkey, SUM(acctbal)
FROM customer
WHERE mktsegment = 'building'
GROUP BY nationkey
```

Let q_G be query obtained from q by removing its aggregate expression, that is:


```

SELECT nationkey
FROM customer
WHERE mktsegment = 'building'
GROUP BY nationkey

```

It is easy to see that nation $n1$ is the only consistent answer to q_G on D . Therefore, the range-consistent answer consists of a range of values for $n1$. There are four repairs for the D w.r.t. the key constraint, namely:

$$D_1 = \{ \langle c1, n1, building, 1000 \rangle, \langle c2, n1, building, 500 \rangle, \langle c3, n2, banking, 100 \rangle \}$$

$$D_2 = \{ \langle c1, n1, building, 1000 \rangle, \langle c2, n1, banking, 600 \rangle, \langle c3, n2, banking, 100 \rangle \}$$

$$D_3 = \{ \langle c1, n1, building, 2000 \rangle, \langle c2, n1, building, 500 \rangle, \langle c3, n2, banking, 100 \rangle \}$$

$$D_4 = \{ \langle c1, n1, building, 2000 \rangle, \langle c2, n1, banking, 600 \rangle, \langle c3, n2, banking, 100 \rangle \}$$

The result of q on the repairs is the following:

$$q(D_1) = \{ \langle n1, 1500 \rangle \}$$

$$q(D_2) = \{ \langle n1, 1000 \rangle \}$$

$$q(D_3) = \{ \langle n1, 2500 \rangle \}$$

$$q(D_4) = \{ \langle n1, 2000 \rangle \}$$

Hence, the (range-consistent) answer to q is $\{ \langle n1, 1000, 2500 \rangle \}$, because the sum of the account balances for customers in the building sector and nation $n1$ is:

- between 1000 and 2500, in every repair,
- 1000 in the repair D_2 ,
- 2500 in the repair D_3 .

In the next example we illustrate the rewriting approach.

Example 2.8. We now show the rewriting for the query q of Example 2.7. First, upper and lower bounds for the account balance of each customer are obtained, and finally the sum of the account balances is computed.

The following subquery retrieves the lower and upper bounds for the account balance of the customers which always satisfies the query q_G .

```

UnFilteredCandidates AS (
  SELECT custkey, nationkey,
         MIN(acctbal) AS minBal, MAX(acctbal) AS maxBal
  FROM customer c
  WHERE mktsegment = 'building'
         AND NOT EXISTS ( SELECT *
                          FROM Filter
                          WHERE c.custkey=Filter.custkey)
  GROUP BY custkey, nationkey)

```

The `Filter` subquery has been generated for the query q_G as previously discussed for queries without aggregation or grouping. The filter retrieves the customers which appear in some tuple that does not satisfy q_G . By applying the

filter on D , $c2$ and $c3$ are discarded from the result of `UnFilteredCandidates`. The customer $c1$ is not filtered because its two tuples satisfy the query q_G . In the repairs D_1 and D_2 , the account balance of $c1$ is 1000, whereas in the repairs D_3 and D_4 , the account balance of $c1$ is 2000. Therefore, it contributes a minimum of 1000 and a maximum of 2000. Indeed, the result of `UnFilteredCandidates` on D is $\{ \langle c1, n1, 1000, 2000 \rangle \}$. Notice that the filter is necessary because we would otherwise get the tuple $\langle c2, n1, 500, 500 \rangle$ in the result, which states that customer $c2$ contributes an amount of 500 in every repair. This is not correct, since in the repairs D_2 and D_4 , $c2$ does not satisfy the query q_G and therefore does not contribute to the sum of account balances. Thus, $c2$ contributes a minimum of 0 and a maximum of 500. This is captured with the following query:

```
FilteredCandidates AS (
  SELECT custkey, nationkey,
         0 AS minBal, MAX(acctbal) AS maxBal
  FROM customer c
  WHERE mktsegment = 'building'
        AND EXISTS ( SELECT *
                     FROM Filter
                     WHERE Filter.custkey=c.custkey)
        AND EXISTS ( SELECT *
                     FROM QGCons
                     WHERE QGCons.nationkey=c.nationkey)
  GROUP BY custkey, nationkey)
```

The result of `FilteredCandidates` is $\langle c2, n1, 0, 500 \rangle$. In addition to checking that the customer is filtered, we check that the nation (i.e., the attribute in the group by of the original query) appears in the result of the consistent answers to q_G (denoted as `QGCons` in the query). This is necessary because we do not want to retrieve ranges for the nations that are not consistent answers. Finally, we obtain the range-consistent answers by summing up the lower and upper bounds for each nation in the result of `FilteredCandidates` and `UnfilteredCandidates`, as follows:

```
SELECT nationkey, SUM(minBal), SUM(maxBal)
FROM ( SELECT * FROM FilteredCandidates
      UNION ALL
      SELECT * FROM UnfilteredCandidates)
GROUP BY nationkey
```

In general, the rewriting deals with negative values as well.

2.2.2 Representing all Repairs

Chomicki et al. [26, 27] propose an approach for computing consistent query answers which uses a representation of the integrity constraint violations.

Specifically, projection-free relational algebra queries and denial constraints are considered. In [25] it has been shown that consistent answers to projection-free queries in the presence of denial constraints can be computed in polynomial time (data complexity).

The approach proposed in [26, 27] relies on the notion of *conflict hypergraph* which represents all the integrity violations in a given database. Given a database D and a set IC of denial constraints, the conflict hypergraph for D w.r.t. IC , denoted by $\mathcal{G}_{D,IC}$, is a hypergraph whose set of vertices is the set of tuples in D , whereas the set of edges consists of all the minimal sets of tuples in D violating together a denial constraint in IC .

We first review the polynomial time algorithm presented in [25] for checking the consistency of ground queries in the presence of denial constraints, then we show how to use it to answer projection-free relational algebra queries. We recall that the consistent answer to a ground query q on a database D w.r.t. a set IC of denial constraints is *true* (resp. *false*) if $R \models q$ (resp. $R \not\models q$) for every repair R of D and IC .

Queries are assumed to be in CNF. This assumption does not reduce the generality of the result, because every ground query can be converted to CNF independently of the database, and thus without affecting the data complexity of query evaluation. However, from a practical point of view, CNF conversion may lead to unacceptably complex queries.

A query q is *true* in every repair of a database D if and only if each of the conjunct of q is *true* in every repair of D . So the first step of the algorithm reduces the task of determining whether *true* is the consistent answer to q to answering the same question for every conjunct $\psi : t_1 \vee \dots \vee t_m \vee \neg t_{m+1} \vee \dots \vee \neg t_n$ of q . This can be done by checking whether there exists a repair R in which $\neg\psi$ is *true*. If such a repair is not found for every conjunct of q , then *true* is the consistent answer to q .

A repair in which $\neg\psi$ is *true* contains t_{m+1}, \dots, t_n and does not contain t_1, \dots, t_m . The algorithm selects nondeterministically for every j , $1 \leq j \leq m$, an edge e_j in $\mathcal{G}_{D,IC}$ such that t_j is in e_j , and constructs a set of facts S such that

$$S = \{t_{m+1}, \dots, t_n\} \cup \bigcup_{1 \leq j \leq m, t_j \in D} (e_j - \{t_j\})$$

and there is no edge e of $\mathcal{G}_{D,IC}$ such that $e \subseteq S$. If the construction of S succeeds, then a repair in which $\neg\psi$ is *true* can be built by adding to S new tuples from D until the set is maximal independent. The algorithm needs m nondeterministic steps, a number which is independent of the size of the database (but dependent on q), and in each of its nondeterministic steps selects one possibility from a set whose size is polynomial in the size of the database. So there is an equivalent PTIME deterministic algorithm.

Any relational algebra query q can be translated to a corresponding first-order formula $\Phi_q(X)$ in a standard way. Since we consider only projection-free relational algebra queries, the formula $\Phi_q(X)$ is quantifier-free. To be

able to use the previous algorithm for ground queries, we have to ground this formula, i.e., find an appropriate set of bindings for the variables in the formula. This is done by evaluating an *envelope* query over the database. An envelope query should satisfy two properties: (1) it should return a superset of the set of consistent query answers for every database, and (2) it should be easily constructible from the original query. Suppose that E_q is an envelope query for a query q and let D and IC be a database and a set of denial constraints respectively; then $consistent_{IC}(q, D) = \{t \in E_q(D) \mid D \models_{IC} \Phi_q(t)\}$

If a query does not use the difference operator (and thus is a monotonic expression), the query itself is an envelope query. This may not be the case when the query is not monotonic. In order to obtain an envelope query, two operators F and G are defined by mutual recursion. The operator F defines the envelope by overestimating the set of consistent answers. The auxiliary operator G underestimates the set of consistent answers. They are recursively defined as follows.

$$\begin{aligned}
 F(R) &= R \\
 F(E_1 \cup E_2) &= F(E_1) \cup F(E_2) \\
 F(E_1 \setminus E_2) &= F(E_1) \setminus G(E_2) \\
 F(E_1 \times E_2) &= F(E_1) \times F(E_2) \\
 F(\sigma_c(E)) &= \sigma_c(F(E)) \\
 G(R) &= \Delta_{IC}^R \\
 G(E_1 \cup E_2) &= G(E_1) \cup G(E_2) \\
 G(E_1 \setminus E_2) &= G(E_1) \setminus F(E_2) \\
 G(E_1 \times E_2) &= G(E_1) \times G(E_2) \\
 G(\sigma_c(E)) &= \sigma_c(G(E))
 \end{aligned}$$

where R is a relation, E_1 and E_2 are relational algebra expressions, Δ_{IC}^R is a relational algebra expression which evaluated over R gives the tuples common to all the repair of R w.r.t. IC .

The paper [26] presents some optimizations for the previous algorithm and experimental results.

The technique described above underlies the implementation of the system Hippo [26, 27].

2.2.3 Logic Programs

A major line of work on CQA involves capturing repairs as answer sets of logic programs with negation and disjunction [7, 20, 49, 50]. Such approaches are quite general, being able to handle arbitrary universal constraints and first-order queries. Determining whether an atom is a member of all answer sets of such a logic program is Π_2^P -complete. Therefore, a direct implementation of CQA using a disjunctive logic programming system like DLV [59] is practical only for very small databases. We next present two techniques for computing repairs by means of extended disjunctive logic programs and logic

programs with exceptions, respectively, derived from the given set of integrity constraints.

Computing Consistent Query Answers using Extended Disjunctive Logic Programs

A logical framework for computing repairs and consistent query answers has been proposed in [49, 50]. The proposed technique consists in rewriting integrity constraints into extended disjunctive logic programs, so that repairs and consistent query answers can be derived from the stable models of such programs.

The paper [50] introduces also *repair constraints*, a form of constraints which can be used to specify which repairs are *feasible*, and *prioritized update rules* which allow us to express preferences among repairs. A set of integrity and repair constraints can be rewritten into an extended disjunctive logic program such that feasible repairs corresponds to stable models and vice versa; integrity constraints with prioritized updates can be rewritten into a prioritized disjunctive program [70] so that *preferred repairs* correspond to preferred stable models of the obtained program and vice versa.

Universal integrity constraints are considered. It is assumed that they are written under the following form:

$$\forall X[B_1 \wedge \dots \wedge B_n \wedge \phi \rightarrow A_1 \vee \dots \vee A_m]$$

where the A_i 's and the B_j 's are atoms, ϕ is a conjunction of built-in atoms, X denotes the list of all variables appearing in the B_j 's; variables appearing in the A_i 's and in ϕ also appear in the B_j 's.

Given a universal constraint c of the form above, $dj(c)$ denotes the extended disjunctive rule

$$\neg B'_1 \vee \dots \vee \neg B'_n \vee A'_1 \vee \dots \vee A'_m \leftarrow (B_1 \vee B'_1), \dots, (B_n \vee B'_n), \phi, \\ (\text{not } A_1 \vee \neg A'_1), \dots, (\text{not } A_m \vee \neg A'_m)$$

where C'_i denotes the atom derived from C_i by replacing the predicate symbol p with the new predicate symbol p_u . Observe that *not* and \neg denote respectively negation as failure and classical negation. The semantics of a program with classical negation is defined by considering each negated predicate symbol $\neg p$ as a new predicate symbol syntactically different from p and by adding to the program, for each predicate symbol p with arity n , the constraint $\leftarrow p(x_1, \dots, x_n), \neg p(x_1, \dots, x_n)$. Given a set IC of universal constraints, $LP(IC) = \{dj(c) \mid c \in IC\}$.

Given a database D and a set IC of universal constraints, let M be a stable model of $LP(IC) \cup D$; then

$$\mathcal{R}(M) = (\{p(t) \mid p_u(t) \in M \wedge p(t) \notin D\}, \\ \{p(t) \mid \neg p_u(t) \in M \wedge p(t) \in D\})$$

Intuitively, the first (resp. second) set of $\mathcal{R}(M)$, denoted as $\mathcal{R}(M)^+$ (resp. $\mathcal{R}(M)^-$), corresponds to the set of atoms which should be inserted into (resp. deleted from) the original database in order to obtain a repair. We denote by $\mathcal{R}(M, D)$ the database obtained from D by applying the deletions and the insertions specified by M , that is $\mathcal{R}(M, D) = D \cup \mathcal{R}(M)^+ - \mathcal{R}(M)^-$.

In [50], it has been shown that for every repair S of D w.r.t. IC there exists a stable model M of $LP(IC) \cup D$ s.t. $\mathcal{R}(M, D) = S$, and every stable model M of $LP(IC) \cup D$ is s.t. $\mathcal{R}(M, D)$ is a repair of D w.r.t. IC .

Example 2.9. Consider the database $D = \{p(a), p(b), q(a), q(c)\}$ and the inclusion dependency $(\forall x) [p(x) \rightarrow q(x)]$. Clearly, D is inconsistent as it does not satisfy $p(b) \rightarrow q(b)$. The repairs of D w.r.t. the inclusion dependency are $R_1 = \{p(a), p(b), q(a), q(c), q(b)\}$ and $R_2 = \{p(a), q(a), q(c)\}$. The rewriting of the integrity constraint produces a program LP consisting of the disjunctive rule:

$$\neg p_u(x) \vee q_u(x) \leftarrow (p(x) \vee p_u(x)), (not\ q(x) \vee \neg q_u(x))$$

which can be rewritten into the simpler form

$$\neg p_u(x) \vee q_u(x) \leftarrow p(x), not\ q(x)$$

since the predicates p_u and $\neg q_u$ do not appear in the head of any rule. The program $LP \cup D$ has two stable models, namely $M_1 = D \cup \{q_u(b)\}$ and $M_2 = D \cup \{\neg p_u(b)\}$. Thus, $\mathcal{R}(M_1, D) = R_1$ and $\mathcal{R}(M_2, D) = R_2$.

A query is a pair $\langle g, P \rangle$ where P is a stratified, non-recursive, normal Datalog program and g , which is called *query goal*, is a predicate symbol defined in P specifying the output relation. The answers to a query q on a database D w.r.t. a set IC of integrity constraints are as follows:

$$\begin{aligned} q(D, IC)^+ &= \{g(t) \mid \forall M \in \mathcal{SM}(LP(IC) \cup D) \ g(t) \in \mathcal{SM}(P \cup \mathcal{R}(M, D))\} \\ q(D, IC)^- &= \{g(t) \mid \nexists M \in \mathcal{SM}(LP(IC) \cup D) \ s.t. \ g(t) \in \mathcal{SM}(P \cup \mathcal{R}(M, D))\} \\ q(D, IC)^u &= \{g(t) \mid \exists M_1, M_2 \in \mathcal{SM}(LP(IC) \cup D) \ s.t. \\ &\quad g(t) \in \mathcal{SM}(P \cup \mathcal{R}(M_1, D)), g(t) \notin \mathcal{SM}(P \cup \mathcal{R}(M_2, D))\} \end{aligned}$$

Example 2.10. Consider the database D and the set IC of integrity constraints of Example 2.9. Moreover, let q be the query $\langle s, P \rangle$, where P consists of the rule $s(x) \leftarrow p(x), q(x)$. Then, $q(D, IC)^+ = \{s(a)\}$, $q(D, IC)^u = \{s(b)\}$, and $q(D, IC)^-$ contains those atoms which are neither in $q(D, IC)^+$ nor in $q(D, IC)^u$.

The proposed technique is general but expensive. In [50] it has been shown that, given a database D , a query q , a set IC of functional dependencies and a ground atom t , then checking whether

- $t \in q(D, IC)^+$ is co-NP complete,
- $t \in q(D, IC)^-$ is co-NP complete,
- $t \in q(D, IC)^u$ is NP complete

The co-NP hardness for the first case has also been proved in [5, 8] and subsequently in [25]. In [50] tractable cases have been identified as well: the answers to a query of the form $\langle g, \emptyset \rangle$ on a database w.r.t. a set of functional dependencies (or a set of full inclusion dependencies) can be computed in polynomial time. Observe that these tractable cases have been identified also in [4, 25].

Now, *repair constraints*, a special type of constraints which allow us to restrict the number of repairs, are introduced. A repair constraint is a rule of the form

$$\leftarrow up_1(A_1), \dots, up_m(A_m), L_1, \dots, L_n$$

where $up_1, \dots, up_m \in \{insert, delete\}$, A_1, \dots, A_m are atoms and L_1, \dots, L_n are literals. Informally, the semantics of a repair constraint is as follows: if the conjunction L_1, \dots, L_n is true in a repair R then at least one of the updates $up_i(A_i)$ which have led to R must be false.

Given a database D , a set IC of integrity constraints and a set RC of repair constraints, a repair R of D w.r.t. IC satisfies RC (written $R \models RC$) if for each $\leftarrow insert(A_1), \dots, insert(A_k), delete(A_{k+1}), \dots, delete(A_m), L_1, \dots, L_n$ in RC then

- there is some A_i , $1 \leq i \leq k$, which is not in $R - D$ (that is, A_i has not been inserted into D in order to get R), or
- there is some A_i , $k + 1 \leq i \leq m$, which is not in $D - R$ (that is, A_i has not been deleted from D in order to get R), or
- there is some L_i which is false in R .

The repair R is said to be *feasible* if it satisfies RC .

Example 2.11. Consider the database D and the inclusion dependency of Example 2.9. The repair constraints

$$\begin{aligned} &\leftarrow delete(q(X)) \\ &\leftarrow insert(q(X)) \end{aligned}$$

state that the relation q cannot be modified. There is only one repair which satisfies the repair constraints above, namely $R_2 = \{p(a), q(a), q(c)\}$ because it is obtained by deleting the fact $p(b)$ from D ; the other repair $R_1 = \{p(a), p(b), q(a), q(c), q(b)\}$, obtained by inserting the fact $q(b)$, is not feasible.

The formal semantics of databases with both integrity and repair constraints is given by rewriting the repair constraints into extended rules with empty heads. In particular, the sets of integrity constraints IC and repair constraints RC are rewritten into an extended disjunctive program LP . Each stable model of $LP \cup D$ can be used to generate a feasible repair for the database.

Given a repair constraint r of the form

$$\leftarrow \text{insert}(A_1), \dots, \text{insert}(A_k), \text{delete}(A_{k+1}), \dots, \text{delete}(A_m), \\ B_1, \dots, B_l, \text{not } B_{l+1}, \dots, \text{not } B_n, \varphi$$

where the A_i 's and the B_i 's are base atoms, and φ is a conjunction of built-in atoms, $dj(r)$ denotes the rule (with empty head)

$$\leftarrow A'_1, \dots, A'_k, \neg A'_{k+1}, \dots, \neg A'_m, \\ ((B_1, \text{not } \neg B'_1) \vee B'_1), \dots, ((B_l, \text{not } \neg B'_l) \vee B'_l), \\ ((\text{not } B_{l+1}, \text{not } B'_{l+1}) \vee \neg B'_{l+1}), \dots, ((\text{not } B_n, \text{not } B'_n) \vee \neg B'_n), \varphi$$

where C'_i is derived from C by replacing the predicate symbol, say p , with p_u . Given a set RC of repair constraints, then $LP(RC) = \{ dj(r) \mid r \in RC \}$. $LP(IC, RC)$ denotes the set $LP(IC) \cup LP(RC)$.

In order to satisfy the rule $dj(r)$

- some atom A'_i ($1 \leq i \leq k$) must be false (i.e. A_i is not inserted into the database), or
- some atom $\neg A'_j$ ($k+1 \leq j \leq m$) must be false (i.e. A_j is not deleted from the database), or
- some formula $((B_i, \text{not } \neg B'_i) \vee B'_i)$ ($1 \leq i \leq l$) must be false (i.e. the atom B_i is false in the repair), or
- some formula $((\text{not } B_i, \text{not } B'_i) \vee \neg B'_i)$ ($l+1 \leq j \leq n$) must be false (i.e. the atom B_i is true in the repair), or
- the conjunction of built-in literals φ must be false.

Observe that the formula $(B_i, \text{not } \neg B'_i) \vee B'_i$ states that the atom B_i is true in the repair as either it is in the original database and is not deleted or it is inserted into the original database. Likewise, the formula $(\text{not } B_i, \text{not } B'_i) \vee \neg B'_i$ states that the atom B_i is false in the repair as either it is not in the original database and is not inserted or it is deleted from the original database.

Example 2.12. Consider the repair constraint:

$$\leftarrow \text{insert}(p(a)), \text{delete}(p(b)), r(c), \text{not } q(b)$$

The derived rule is

$$\leftarrow p_u(a), \neg p_u(b), ((r(c), \text{not } \neg r_u(c)) \vee r_u(c)), \\ ((\text{not } q(b), \text{not } q_u(b)) \vee \neg q_u(b))$$

In [50], it has been shown that given a database D , a set IC of universal constraints, a set RC of repair constraints, for every feasible repair S of D w.r.t. IC and RC there exists a stable model M of $LP(IC, RC) \cup D$ s.t. $\mathcal{R}(M, D) = S$, and every stable model M of $LP(IC, RC) \cup D$ is s.t. $\mathcal{R}(M, D)$ is a feasible repair of D w.r.t. IC and RC .

Prioritized update rules are now introduced. They allows us to express preferences among updates and, consequently, among repairs. A *prioritized update rule* is of the form

$$up_1(A) \preceq up_2(B)$$

where $up_1, up_2 \in \{insert, delete\}$, A and B are atoms. Given a set PC of prioritized update rules, we denote by PC^* the reflexive, transitive closure of PC . A set of prioritized update rules PC is said to be *consistent* if there are not two prioritized update rules in PC^* of the form $up_1(A') \preceq up_2(B')$ and $up_2(B'') \preceq up_1(A'')$ such that A' unifies with A'' and B' unifies with B'' . Given a repair R of a database D , $update(R) = \{insert(A) \mid A \in R - D\} \cup \{delete(A) \mid A \in D - R\}$ denotes the set of update atoms used to obtain R from D .

Given a database D , a set IC of integrity constraints, a set RC of repair constraints and a set PC of prioritized update rules, the relation \sqsubseteq is defined over the repairs of D as follows. For any repairs R_1, R_2 and R_3 of D

- $R_1 \sqsubseteq R_1$,
- $R_1 \sqsubseteq R_2$ if
 1. $\exists e_2 \in update(R_2) - update(R_1)$
 $\exists e_1 \in update(R_1) - update(R_2)$
 such that $(e_1 \preceq e_2) \in PC^*$ and
 2. $\nexists e_3 \in update(R_1) - update(R_2)$ such that $(e_2 \preceq e_3) \in PC^*$
- if $R_1 \sqsubseteq R_2$ and $R_2 \sqsubseteq R_3$, then $R_1 \sqsubseteq R_3$.

If $R_1 \sqsubseteq R_2$ we say that R_2 is *preferable* to R_1 . We write $R_1 \sqsubset R_2$ if $R_1 \sqsubseteq R_2$ and $R_1 \neq R_2$. A repair R for D is said to be *preferred* if there is no repair R' for D preferable to R (i.e. such that $R \sqsubset R'$).

A prioritized disjunctive program [70] can be obtained by rewriting integrity and repair constraints into disjunctive rules as it has been shown before and then adding prioritized rules obtained from the rewriting of prioritized update rules. Such a technique is sound and complete, that is for every preferred repair S there exists a preferred stable model M of the prioritized disjunctive program s.t. $\mathcal{R}(M) = S$, and every preferred stable model M of the prioritized disjunctive program is s.t. $\mathcal{R}(M)$ is a preferred repair.

Computing Consistent Query Answers using Logic Programs with Exceptions

Another approach for computing consistent query answers by means of logic programs has been proposed in [6, 7]. Specifically, a generalization of *Logic Program with Exceptions* (LPE) [58], called *Disjunctive Logic Program with Exceptions* (DLPE), is exploited. Given a database and a set of (domain independent) binary universal constraints, the technique allows us to provide a DLPE s.t. there is a one-to-one correspondence between the *e-answers sets* of the constructed program and the repairs. Therefore, the consistent answers to general first-order queries can be obtained asking for the atoms which are true in every e-answer set of the logic program.

We briefly introduce LPEs and DLPEs. Logic programs with exceptions (LPEs) are built with normal extended clauses, that is, with clauses where the (non-disjunctive) head and the body are literals (with classical negation) and weak negation (or negation as failure) may appear in the bodies [44]. Among those clauses, in a LPE there are *positive default rules*, that is clauses with positive heads, whose conclusions can be overridden by conclusions derived from *exception rules*, which are clauses with negative heads. The idea is that exceptions have priority over defaults. To capture this intuition a new semantics is introduced, namely *e-answer sets*.

Example 2.13. We show a logic program with exceptions that cleans a database $r(X, Y)$ from tuples participating in the violation of the functional dependency $X \rightarrow Y$. First, we introduce a new predicate $r'(X, Y)$ that will store the tuples in the clean version of the database. The program consists of the following rules:

1. *Default rule:* $r'(X, Y) \leftarrow r(X, Y)$.
It says that every tuple (X, Y) passes from r to r' .
2. *Negative exception rule:* $\neg r(X, Y) \leftarrow r(X, Y), r(X, Z), \text{not } Y = Z$.¹
It says that tuples (X, Y) in r where X is associated to different values are not accepted in the clean table.
3. *Facts:* the contents of r plus $X = X \leftarrow$.

Intuitively, rule 2 should have a priority over rule 1.

The semantics of the program should take the priorities into account; they should be reflected in the intended models of the program. The semantics of an LPE is as follows. Let Π be a ground LPE and S be a set of ground literals. A set of ground rules Π^S is generated as follows:

- (a) Delete every rule in Π containing *not* L in the body, with $L \in S$.
- (b) Delete from the remaining clauses every condition *not* L in the body, when $L \notin S$.
- (c) Delete every rule having a positive conclusion L with $\neg L \in S$.

The result is a ground extended logic program without *not*. We say that S is an *e-answer set* of the original program if it is an answer set of Π^S . Observe that (a) and (b) above are as in the answer sets semantics for extended logic programs [43], whereas (c) takes exceptions into account.

In order to specify database repairs, LPEs are extended to accommodate also *negative defaults*, i.e. defaults with negative conclusions that can be overridden by *positive exceptions*, and *extended disjunctive exceptions*, i.e. rules of the form

¹ Observe that *not* and \neg denote respectively negation as failure and classical negation. The semantics of a program with classical negation is defined by considering each negated predicate symbol $\neg p$ as a new predicate symbol syntactically different from p and by adding to the program, for each predicate symbol p with arity n , the constraint $\leftarrow p(x_1, \dots, x_n), \neg p(x_1, \dots, x_n)$.

$$L_1 \vee \dots \vee L_k \leftarrow L_{k+1}, \dots, L_r, \text{not } L_{r+1}, \dots, \text{not } L_n$$

where the L_i 's are literals. The e-answer semantics is extended as follows. The program Π^S is obtained by applying (a) and (b) as before whereas a new version of the rule (c) has to be applied:

- (c') delete every (positive) default having a positive conclusion L , with $\neg L \in S$ and every (negative) default having a negative conclusion $\neg L$, with $L \in S$.

Now the result is a ground disjunctive logic program without *not*. If the candidate set of literals S belongs to the answer set of Π^S , namely the set of minimal models of program Π^S , then we say that S is an *e-answer set*. The so obtained program can be transformed into a disjunctive extended logic program with answer set semantics; the latter can be transformed in turn into a disjunctive program with stable model semantics.

We now present the approach of [6, 7]. Binary universal constraints written in the standard format are considered, that is constraints of the form

$$\forall X_1, X_2 [p_1(X_1) \vee p_2(X_2) \vee \phi(X_1, X_2)]$$

where X_1, X_2 are tuples of variables and ϕ is a formula containing only built-in predicates. There are three possibilities for binary constraints in terms of sign of literals in them, namely the universal closures of:

- (a) $p_1(X_1) \vee p_2(X_2) \vee \phi(X_1, X_2)$
- (b) $p_1(X_1) \vee \neg p_2(X_2) \vee \phi(X_1, X_2)$
- (c) $\neg p_1(X_1) \vee \neg p_2(X_2) \vee \phi(X_1, X_2)$

As said before the approach consists in deriving a DLPE Π_{BC} from a set BC of binary universal constraints so that Π_{BC} can be used to compute consistent query answers. For each predicate symbol p that appears in some integrity constraint, a new predicate symbol p' representing the repaired version of p is introduced; p' contains the tuples corresponding to p in a repair of the original database.

Moreover, Π_{BC} is obtained by introducing:

1. **Persistence Defaults.** For each base predicate p , the following persistence defaults are introduced:

$$p'(X) \leftarrow p(X) \tag{2.1}$$

$$\neg p'(X) \leftarrow \text{not } p(X) \tag{2.2}$$

The defaults say that all data persist from the original relations to their repaired versions. The *positive defaults* (rules of type 2.1) will be subject to *negative exceptions*, whereas the *negative defaults* (rules of type 2.2) will be subject to *positive exceptions*.

2. **Stabilizing Exceptions.** For each constraint of the form (a), i.e. $p_1(X_1) \vee p_2(X_2) \vee \phi(X_1, X_2)$, the following pair of *positive exception clauses* is introduced:

$$\begin{aligned} p'_1(X_1) &\leftarrow \neg p'_2(X_2), \varphi(X_1, X_2) \\ p'_2(X_2) &\leftarrow \neg p'_1(X_1), \varphi(X_1, X_2) \end{aligned}$$

where φ is the negation of ϕ .

Similarly, for each constraint of the form (b), i.e. $p_1(X_1) \vee \neg p_2(X_2) \vee \phi(X_1, X_2)$, the following clauses are introduced

$$\begin{aligned} p'_1(X_1) &\leftarrow p'_2(X_2), \varphi(X_1, X_2) \\ \neg p'_2(X_2) &\leftarrow \neg p'_1(X_1), \varphi(X_1, X_2) \end{aligned}$$

Finally, for each constraint of the form (c), i.e. $\neg p_1(X_1) \vee \neg p_2(X_2) \vee \phi(X_1, X_2)$, the pair of *negative exception clauses* is introduced

$$\begin{aligned} \neg p'_1(X_1) &\leftarrow p'_2(X_2), \varphi(X_1, X_2) \\ \neg p'_2(X_2) &\leftarrow p'_1(X_1), \varphi(X_1, X_2) \end{aligned}$$

These exceptions may override the persistence stated in the defaults above. The meaning of the *stabilizing exceptions* is to make the integrity constraints be satisfied by the new predicates p'_i . These exceptions are necessary, but not sufficient to ensure that the changes, that the original predicate should be subject to in order to restore consistency, are propagated to the new predicates.

3. **Triggering Exceptions.** For each constraint of the form (a),(b) or (c) the following *disjunctive exception clause* is introduced, respectively:

$$(a) \quad p'_1(X_1) \vee p'_2(X_2) \leftarrow \text{not } p_1(X_1), \text{not } p_2(X_2), \varphi(X_1, X_2)$$

(b)

$$p'_1(X_1) \vee \neg p'_2(X_2) \leftarrow \text{not } p_1(X_1), p_2(X_2), \varphi(X_1, X_2)$$

(c)

$$\neg p'_1(X_1) \vee \neg p'_2(X_2) \leftarrow p_1(X_1), p_2(X_2), \varphi(X_1, X_2)$$

These rules are necessary as a first step towards the repair of the original database. They trigger the first changes, from the p_i 's to the p'_i 's; next the stabilizing exceptions propagate all required changes.

Given a database D and a set BC of domain independent binary universal constraints, there is a one-to-one correspondence between the e-answer sets of $\Pi_{BC} \cup D$ and the set of repairs for D w.r.t. BC . The consistent answers to a query q over D w.r.t. BC are those atoms which are in every e-answer set of $q \cup \Pi_{BC} \cup D$, that is those atoms which are *true* under the *cautious* or *skeptical* answer set semantics.

This approach is very general because it applies to arbitrary first-order queries. However, the systems computing answer sets work typically by grounding the logic program. In the database context, this may lead to huge ground programs and be impractical. In the general case, computing the stable model semantics for disjunctive programs is Π_2^P -complete in the size of the ground program. The deductive database system DLV [59] can be used for computing repairs and consistent query answers.

2.3 Computing CQAs: Computational Complexity

As we observed before the definition of consistent query answer does not yield a practical method for computing consistent answers. Indeed, in general the problem is intractable. The table below summarizes the complexity results obtained for different classes of queries (here we refer to the relational algebra) and constraints. The data complexity assumption [75] is adopted, that is the complexity of the problem is measured as a function of the number of tuples in a database; the query and the integrity constraints are considered fixed.

	<i>Primary keys</i>	<i>Arbitrary keys</i>	<i>Denial</i>	<i>Universal</i>
$\sigma \times -$	P [4]	P[4]	P [25]	P (binary) [4] Π_2^P -complete [71]
$\sigma \times - \cup$	P [25]	P [25]	P [25]	Π_2^P -complete [71]
$\sigma \pi$	P [25]	co-NPC [25]	co-NPC [25]	Π_2^P -complete [71]
$\sigma \pi \times$	co-NPC [25] P (C_{forest}) [42]	co-NPC [25]	co-NPC [25]	Π_2^P -complete [71]
$\sigma \pi \times - \cup$	co-NPC [25]	co-NPC [25]	co-NPC [25]	Π_2^P -complete [71]

Fig. 2.1. Complexity of CQA: relational algebra

Observe that in the presence of denial constraints the problem of computing consistent query answers is in PTIME as long as the projection is not used [4, 25]. By using the projection operator, the problem becomes in general intractable [25]. However, in the presence of primary keys only, a particular class of conjunctive queries, called \mathcal{C}_{forest} , for which the problem is in PTIME has been identified [42] (see Section 2.2.1). Finally, observe that in the presence of universal integrity constraints, the problem is in general Π_2^P -complete [71], although for binary universal constraints and queries not using the projection and the union operators the problem is in PTIME [4].

2.4 Variants of CQA

After the paper [4], several works varying the notions of repair and consistent query answer have been proposed. We first survey those works adopting a notion of repair which differs from the original one.

For denial constraints, integrity violations can only be removed by deleting tuples, so all the repairs are subsets of the given database. In the presence of general universal or referential integrity constraints, violations can also be removed by adding tuples. Allowing tuple insertions makes sense if the information in the source database may be incomplete (e.g. in data integration applications). On the other hand, if the data in the database is complete but possibly incorrect, as in data warehousing applications, it is natural to consider only repairs constructed using tuple deletions.

The above considerations have led to the definition of new classes of repairs:

- *D-repairs*, constructed using a minimal set of deletions [25],
- *I-repairs*, constructed using a minimal set of deletions and some, not necessarily minimal set of insertions [19].

These classes of repairs lead to different notions of consistent query answers.

If we consider D-repairs, given a set F of primary key functional dependencies and a set IN of foreign-key constraints, every repair of a database w.r.t. $F \cup IN$ may be obtained as a repair of the single D-repair of the database w.r.t. IN (this unique repair can be obtained by deleting the tuples violating the foreign-key constraints). Then, one can adapt any polynomial-time method for CQA w.r.t. primary key constraints, for example [42], to compute consistent query answers w.r.t. any set of primary key and foreign-key constraints in polynomial time. However, in more general settings, the interactions between functional and inclusion dependencies get complex, and CQA become quickly intractable [25].

As for I-repairs, in [19] it has been shown that for such repairs in the presence of primary key functional dependencies and arbitrary inclusion dependencies CQA becomes undecidable. A class of inclusion dependencies, called *non-key-conflicting*, is presented, for which the interaction between functional and

inclusion dependencies is limited and consequently CQA is co-NP-complete. In the same setting, by considering the original definition of repair, the problem of consistent query answering is undecidable in the general case whereas is Π_2^P -complete for non-key-conflicting inclusion dependencies.

Furfaro et al. [47, 38] adopt a notion of repair which extends the original one by allowing tuples to be made undefined (in addition to be inserted and deleted). The proposed framework allows us to compute a sound and incomplete set of consistent query answers (according to the original definition) in polynomial time. The work is presented in Chapter 6.

The original notion of repair has been criticized as *too coarse-grained*: deleting a tuple to remove an integrity violation potentially eliminates useful information in that tuple. More fine-grained methods seek to define repairs by minimizing attribute modifications [12, 14, 48, 76]. In particular, Bertossi et al. [12] and Bohannon et al. [14] use various notions of numerical distance between tuples. In both cases the existence of a repair within a given distance of the original database instance turns out to be NP-complete. To achieve tractability the former proposes approximation, and the latter heuristics. Wijsen [77] has shown how to combine tuple- and attribute-based repairs in a single framework. To achieve the effect of attribute-based repairing, his approach decomposes an inconsistent relation using a lossless-join decomposition and subsequently joins the obtained projections. PJ-repairs are defined to be the repairs (according to the original definition) of the resulting relation.

A repairing strategy based on value-modifications has been adopted also in [48], which is presented in Chapter 7.

Now we briefly look at the proposed variants of the notion of consistent query answer.

In order to provide more informative query answers to aggregation queries, Arenas et al. [8] propose to return the minimal interval containing the set of the values of the aggregate function obtained in some repair. The paper [8] contains a detailed analysis of the data complexity of computing interval answers in the presence of functional dependencies, and showed the influence of the cardinality $|F|$ of the given set of functional dependencies F .

The notion of repair and consistent query answer has been generalized to the context of probabilistic databases by Andritsos et al. [2]. In such databases probabilities are associated with individual tuples. The considered constraints consist of one (primary) key per relation. Then the probabilities of the conflicting tuples sum up to one. A repair also has an associated probability, which is the product of the probabilities of the tuples belonging to the repair. There is a natural way to compute the probability of an answer: sum up the probabilities of the repairs in which the answer appears in the query result. Such answers, with the associated probabilities, are called clean answers. Clearly, consistent answers are those clean answers that have probability one. In the same work, a way to compute clean answers through query rewriting is also presented. The method applies to a class of conjunctive queries closely related to \mathcal{C}_{forest} (see Section 2.2.1).

Greco and Molinaro [48] propose a notion of *probabilistic query answer* over inconsistent databases, that is query answers are tuples associated with probabilities. The probability of a tuple depends on the number of repairs from which the answer tuple is obtained. A technique for computing probabilistic answers, relying on probabilistic databases, is presented. Such a technique allows us to compute approximate probabilistic answers in polynomial time. This work is presented in Chapter 7.

2.5 Discussion

In this chapter we have surveyed the literature of consistent query answering. First, we have presented the main techniques for computing CQAs. Specifically, we have classified them into three approaches: query rewriting [4, 41, 42, 39, 40], compact representations of repairs [26, 27], and logic programs [7, 20, 49, 50]. The first two approaches allow us to compute consistent query answers in polynomial time, but have limited applicability; the last approach is applicable to very general queries and constraints, but leads to high complexity. Next, we have proposed complexity results on computing consistent query answers: here the main remark is that in several cases for which the problem is in PTIME, the projection operator leads to intractability. Finally, we have briefly presented the main works in the literature where the notions of repair and consistent query answer differ from the original ones proposed in [4].

Prioritized Active Integrity Constraints

This chapter presents a logical framework wherein constraints and preferences are used for database integrity maintenance and querying. The proposed approach is based on the use of a special type of integrity constraints, called *Prioritized Active Integrity Constraints* (PAICs), whose body defines an integrity constraint, whereas the head contains a set of partially ordered actions, which should be performed, if the body constraint is not satisfied, to make consistent the database. Therefore, a preference relation among repairs is introduced on the base of the (partially ordered) actions specified in the head of PAICs. On the base of the preference relation, a set of preferred repairs is identified and preferred query answers are derived by considering only preferred repairs. We show some desirable properties of preferred repairs which hold in our framework. We also shows how PAICs can be rewritten into disjunctive Datalog programs so that repairs can be obtained from the computation of stable models.

3.1 Introduction

The motivation of this work stems from the observation that as an inconsistent database can be repaired in different ways, it is natural to express preferences among the possible actions which make the database consistent.

Example 3.1. Consider the database schema consisting of the relation schemas $emp(Name, Dept)$ and $dept(Name)$ with a referential integrity constraint stating that a department appearing in the relation emp must occur in the relation $dept$ too. This constraint can be defined through the first order formula:

$$\forall(E, D) [emp(E, D) \rightarrow dept(D)]$$

Consider now the inconsistent instance $D = \{emp(john, cs), emp(john, deis), dept(deis)\}$. The database can be repaired either by inserting the missing

fact $dept(cs)$ or by deleting the fact $emp(john, cs)$. In this scenario, suppose that the insertion of a missing department is preferable to the deletion of an existing employee. This preference can be expressed by means of the following prioritized active integrity constraint:

$$\forall(E, D) [emp(E, D), not dept(D) \rightarrow +dept(D) \succ -emp(E, D)]$$

Then, the repaired database obtained by inserting the fact $dept(cs)$ is preferable to the repaired database obtained by deleting the fact $emp(john, cs)$ and is the unique preferred repaired database.

The novelty of the presented approach consists in the formalization of *Prioritized Active Integrity Constraints (PAICs)*, a flexible and easy mechanism for specifying the preference among updates. More specifically, prioritized active integrity constraints are constraints which allow us to define the actions to be performed if a constraint is violated and also to introduce a partial order on the actions. We show that databases with universally quantified prioritized constraints admit preferred repairs and (preferred) consistent answers if the set of constraints is satisfiable. We also present how the computation of repairs and consistent answers can be done by rewriting constraints into a disjunctive Datalog program and computing the stable models of the target program; every repair can be obtained from a stable model of the Datalog program and vice versa.

The chapter is organized as follows. Section 3.2 present syntax and semantics of prioritized active integrity constraints. Section 3.3 presents a technique for deriving a disjunctive Datalog program from a set of PAICs in such a way that the stable models of the obtained program correspond to *founded* repairs. Finally, in Section 3.4 we discuss related works and draw conclusions.

3.2 Syntax and Semantics of PAICs

In this section an extension of integrity constraints, which allows us to specify for each constraint the actions to be performed to satisfy it and preferences between them, is presented. The actions are defined by means of insertions and deletions. First we present the syntax of PAICs and next their semantics.

Syntax

An update atom is of the form $+a(X)$ or $-a(X)$, where $a(X)$ is a standard atom. A ground atom $+a(t)$ states that $a(t)$ will be inserted into the database, whereas a ground atom $-a(t)$ states that $a(t)$ will be deleted from the database. Given an update atom $+a(X)$ (resp. $-a(X)$), $Comp(+a(X)) = not a(X)$ (resp. $Comp(-a(X)) = a(X)$) denotes the complementary literal of the update atom $+a(X)$ (resp. $-a(X)$). Given a set Up of ground

update atoms, the following sets are defined: $Up^+ = \{a(t) \mid +a(t) \in Up\}$, $Up^- = \{a(t) \mid -a(t) \in Up\}$ and $Comp(Up) = \{Comp(\pm a(t)) \mid \pm a(t) \in Up\}$. Up is said to be *consistent* if it does not contain two update atom $+a(t)$ and $-a(t)$ (i.e. if $Up^+ \cap Up^- = \emptyset$). Given a database D and a consistent set of update atoms Up , $Up(D)$ denotes the updated database $D \cup Up^+ - Up^-$.

In this chapter, a minimal set of updates which leads a database to a consistent state will be called *repair* (it is precisely defined below); a *repaired database* is the result of applying a repair over the original database.

Definition 3.1. Given a database D and a set of integrity constraints IC , a *repair* for $\langle D, IC \rangle$ is a consistent set of update atoms R such that 1) $R(D) \models IC$ and 2) there is no consistent set of update atoms $Up \subset R$ such that $Up(D) \models IC$.

Given a database D and a set of integrity constraints IC , the set of all repairs (resp. repaired databases) for $\langle D, IC \rangle$ is denoted as $repairs(D, IC)$ (resp. $repaired(D, IC)$).

Definition 3.2. Given a database D and a set of integrity constraints IC , an atom A is *true* (resp. *false*) with respect to IC if A belongs to all repaired databases (resp. there is no repaired database containing A). The atoms which are neither true nor false are *undefined*.

Definition 3.3. A (universally quantified) *Active Integrity Constraint* (AIC) is of the form:

$$(\forall X) \left[\bigwedge_{j=1}^m b_j(X_j), \bigwedge_{j=m+1}^n \text{not } b_j(X_j), \varphi(X_0) \rightarrow \bigvee_{i=1}^p \pm a_i(Y_i) \right] \quad (3.1)$$

where $X = \bigcup_{j=1}^m X_j$, $X_i \subseteq X$ for $i \in [0, \dots, n]$ and $Y_i \subseteq X$ for $i \in [1, \dots, p]$.

In the above definition the conditions $X = \bigcup_{j=1}^m X_j$, $X_i \subseteq X$ for $i \in [0, \dots, n]$ and $Y_i \subseteq X$ for $i \in [1, \dots, p]$ guarantee that variables are range restricted.

Active integrity constraints contain in the head the actions to be performed if the constraint defined in the body is not satisfied.

Given an AIC r of the form (3.1) $St(r)$ denotes the standard constraint:

$$(\forall X) \left[\bigwedge_{j=1}^m b_j(X_j), \bigwedge_{j=m+1}^n \text{not } b_j(X_j), \varphi(X_0) \rightarrow \right]$$

derived from r by removing the head update atoms. Moreover, for a set of active integrity constraints IC , $St(IC)$ denotes the corresponding set of standard integrity constraints, i.e. $St(IC) = \{St(r) \mid r \in IC\}$.

Definition 3.4. A (universally quantified) *Prioritized Active Integrity Constraint* (PAIC) is of the form:

$$(\forall X) \left[\bigwedge_{j=1}^m b_j(X_j), \bigwedge_{j=m+1}^n \text{not } b_j(X_j), \varphi(X_0) \rightarrow \bigvee_{i=1}^{p_1} \pm a_i^1(Y_i^1) \succ \dots \succ \bigvee_{i=1}^{p_k} \pm a_i^k(Y_i^k) \right] \quad (3.2)$$

where $X = \bigcup_{j=1}^m X_j$, $X_i \subseteq X$ for $i \in [0, \dots, n]$ and $Y_i^j \subseteq X$ for $j \in [1, \dots, k]$ and $i \in [1, \dots, p_j]$.

Prioritized active integrity constraints contain in the head the actions to be performed if the constraint defined in the body is not satisfied and express preferences among them.

Intuitively, the meaning of $\bigvee_{i=1}^{p_1} \pm a_i^1(Y_i^1) \succ \bigvee_{i=1}^{p_2} \pm a_i^2(Y_i^2)$ is that the actions $\pm a_1^1(Y_1^1), \dots, \pm a_{p_1}^1(Y_{p_1}^1)$ are preferable to the actions $\pm a_1^2(Y_1^2), \dots, \pm a_{p_2}^2(Y_{p_2}^2)$.

Given a (P)AIC $r = (\forall X)[\Phi \rightarrow \Psi]$, Φ is called body of r (denoted by $Body(r)$), whereas Ψ is called head of r (denoted by $Head(r)$).

Definition 3.5. A (prioritized) active integrity constraint is said to be in *canonical* form if for each update literal $\pm a(X)$ appearing in the head, a literal $Comp(\pm a(X))$ also appears in the body. A set of (prioritized) active integrity constraints is said to be canonical if all constraints are in canonical form.

In the rest of the chapter, (universally quantified) prioritized active integrity constraints in canonical form are considered. The motivation for restricting our attention to canonical AICs is due to the fact that in [22, 23] it has been shown that for every ground AIC r , every head update atom $\pm A$ such that $Comp(\pm A) \notin Body(r)$ is useless and can be deleted.

Semantics

In the following firstly the definition concerning the truth value of ground atoms and ground update atoms with respect to a database D and a consistent set of update atoms Up is given, then the formal definition of founded and preferred repair is provided.

Definition 3.6. Given a database D and a consistent set of update atoms Up , the truth value of

- a positive ground literal $a(t)$ is *true* w.r.t. (D, Up) if $a(t) \in Up(D)$,
- a negative ground literal $\text{not } a(t)$ is *true* w.r.t. (D, Up) if $a(t) \notin Up(D)$,
- a ground update atom $\pm a(t)$ is *true* w.r.t. (D, Up) if $\pm a(t) \in Up$,
- built-in atoms, conjunctions and disjunctions of literals is given in the standard way,
- a ground AIC $\phi \rightarrow \psi$ is *true* w.r.t. (D, Up) if ϕ is *false* w.r.t. (D, Up) .

Definition 3.7. Let D be a database, IC a set of AICs and R a repair for $\langle D, IC \rangle$.

- A ground update atom $\pm a(t) \in R$ is *founded* if there exists $r \in \text{ground}(IC)$ s.t. $\pm a(t)$ appears in $\text{Head}(r)$ and $\text{Body}(r)$ is *true* w.r.t. $(D, R - \{\pm a(t)\})$. We say that $\pm a(t)$ is *supported* by r w.r.t. R .
- A ground rule $r \in \text{ground}(IC)$ is *applied* w.r.t. (D, R) if there exists $\pm a(t) \in R$ s.t. $\pm a(t)$ appears in $\text{Head}(r)$ and $\text{Body}(r)$ is *true* w.r.t. $(D, R - \{\pm a(t)\})$. We say that r *supports* $\pm a(t)$ w.r.t. R .
- R is *founded* if all its atoms are *founded*.
- R is *unfounded* if it is not founded.

The set of founded update atoms in R with respect to $\langle D, IC \rangle$ is denoted as $\text{Founded}(R, D, IC)$, whereas $\text{Unfounded}(R, D, IC) = R - \text{Founded}(R, D, IC)$. Thus, update atoms of founded repairs are inferable by means of AICs. Given a database D and a set of AICs IC , $\text{repairs}_f(D, IC)$ (resp. $\text{repairs}(D, IC)$) denotes the set of founded repairs (resp. all the repairs) for $\langle D, IC \rangle$. Clearly, the set of founded repairs is contained in the set of repairs ($\text{repairs}_f(D, IC) \subseteq \text{repairs}(D, \text{St}(IC))$).

Example 3.2. Consider the following set of AICs IC :

$$\forall(E, P, D)[\text{mgr}(E, P), \text{prj}(P, D), \text{not emp}(E, D) \rightarrow +\text{emp}(E, D)]$$

$$\forall(E, D_1, D_2)[\text{emp}(E, D_1), \text{emp}(E, D_2), D_1 \neq D_2 \rightarrow -\text{emp}(E, D_1) \vee -\text{emp}(E, D_2)]$$

The first constraint states that every manager E of a project P carried out by a department D must be an employee of D , whereas the second one says that every employee must be in only one department. Consider now the database $D = \{\text{mgr}(e_1, p_1), \text{prj}(p_1, d_1), \text{emp}(e_1, d_2)\}$. There are three repairs for D : $R_1 = \{-\text{mgr}(e_1, p_1)\}$, $R_2 = \{-\text{prj}(p_1, d_1)\}$ and $R_3 = \{+\text{emp}(e_1, d_1), -\text{emp}(e_1, d_2)\}$. R_3 is the only founded repair as only the update atoms $+\text{emp}(e_1, d_1)$ and $-\text{emp}(e_1, d_2)$ are derivable from IC .

Definition 3.8. Let c be a PAIC and IC a set of PAICs, then

- $\mathcal{AC}(c)$ denotes the active constraint derived from c by replacing symbol \succ with \vee . Moreover, $\mathcal{AC}(IC) = \{\mathcal{AC}(c) \mid c \in IC\}$.
- $\mathcal{SC}(c)$ denotes the standard constraint derived from c by deleting the update atoms appearing in the head. Moreover, $\mathcal{SC}(IC) = \{\mathcal{SC}(c) \mid c \in IC\}$ (i.e. $\mathcal{SC}(IC) = \text{St}(\mathcal{AC}(IC))$).
- $\mathcal{CC}(c)$ denotes the active constraint derived from $\mathcal{SC}(c)$ by inserting an update atom $\pm a(X)$ in the head if $\text{Comp}(\pm a(X))$ appears in the body of c . Moreover, $\mathcal{CC}(IC) = \{\mathcal{CC}(c) \mid c \in IC\}$.

Example 3.3. Consider the following set of prioritized active integrity constraints IC :

$$\begin{array}{l} c, \text{not } a, \text{not } b \rightarrow +a \succ +b \succ -c \\ c, \text{not } d \quad \quad \quad \rightarrow -c \end{array}$$

The following constraints can be derived:

- $\mathcal{AC}(IC)$ consists of the active constraints

$$\begin{aligned} c, \text{ not } a, \text{ not } b &\rightarrow +a \vee +b \vee -c \\ c, \text{ not } d &\rightarrow -c \end{aligned}$$

- $\mathcal{SC}(IC)$ consists of the standard constraints

$$\begin{aligned} c, \text{ not } a, \text{ not } b &\rightarrow \\ c, \text{ not } d &\rightarrow \end{aligned}$$

- $\mathcal{CC}(IC)$ consists of the active constraints

$$\begin{aligned} c, \text{ not } a, \text{ not } b &\rightarrow +a \vee +b \vee -c \\ c, \text{ not } d &\rightarrow -c \vee +d \end{aligned}$$

Given a database D and a set of PAICs IC , the set of repairs (resp. founded repairs) for $\langle D, IC \rangle$ is denoted by $repairs(D, IC)$ (resp. $repairs_f(D, IC)$).

Fact 3.1 Given a database D and a set of PAICs IC

- $repairs(D, IC) = repairs(D, \mathcal{AC}(IC)) = repairs(D, \mathcal{SC}(IC))$
- $repairs_f(D, IC) = repairs_f(D, \mathcal{AC}(IC))$ □

The above fact states that the repairs for a database D and a set of PAICs IC can be derived by considering the corresponding active (resp. standard) integrity constraints $\mathcal{AC}(IC)$ (resp. $\mathcal{SC}(IC)$), whereas founded repairs can be derived by considering active constraints $\mathcal{AC}(IC)$, obtained by replacing symbol \succ with \vee in the head of prioritized active integrity constraints.

Definition 3.9 (Preferences between repairs). Let D be a database and IC a set of PAICs. For any repairs R_1, R_2 and R_3 in $repairs(D, IC)$, we say that:

- $R_1 \sqsupseteq R_1$.
- $R_1 \sqsupseteq R_2$ if:
 1. $R_1 \in repairs_f(D, IC)$ and $R_2 \notin repairs_f(D, IC)$, or
 2. a) $R_1, R_2 \in repairs_f(D, IC)$ or $R_1, R_2 \notin repairs_f(D, IC)$ and
 - b) there are two update atoms $\pm a(t) \in R_1$ and $\pm b(u) \in R_2$ and a (ground) prioritized active integrity constraint c such that
 - (i) $head(c) = \dots \pm a(t) \dots \succ \dots \pm b(u) \dots$ and
 - (ii) c supports $\pm a(t)$ w.r.t. R_1 and $\pm b(u)$ w.r.t. R_2 .
- If $R_1 \sqsupseteq R_2$ and $R_2 \sqsupseteq R_3$, then $R_1 \sqsupseteq R_3$.

If $R_1 \sqsupseteq R_2$, then R_1 is *preferable to* R_2 . Moreover, if $R_1 \sqsupseteq R_2$ and $R_2 \not\sqsupseteq R_1$, then $R_1 \sqsupset R_2$. A repair R is a *preferred* repair if there is no repair R' such that $R' \sqsupset R$.

The set of preferred repairs for a database D and a set of prioritized active integrity constraints IC is denoted by $repairs_p(D, IC)$.

Example 3.4. Consider the database $D = \{c\}$ and the set of PAICs IC of Example 3.3. $R_1 = \{-c\}$, $R_2 = \{+a, +d\}$ and $R_3 = \{+b, +d\}$ are the three repairs for $\langle D, IC \rangle$; their relation is: $R_1 \sqsupset R_2 \sqsupset R_3$ and the preferred repair is R_1 .

The next theorem states the relation between preferred, founded and general repairs.

Theorem 3.2. Let D be a database and IC a set of PAICs, then

$$\text{repairs}_p(D, IC) \begin{cases} \subseteq \text{repairs}_f(D, IC) & \text{if } \text{repairs}_f(D, IC) \neq \emptyset \\ \subseteq \text{repairs}(D, IC) & \text{if } \text{repairs}_f(D, IC) = \emptyset \end{cases}$$

□

Given a set D of facts and a predicate symbol p , then $D[p]$ denotes the set of facts in D whose predicate symbol is p . Queries are expressed by means of stratified Datalog programs. More formally, a (*Datalog*) query q is a pair (g, P) where g is a predicate symbol, called the *query goal*, and P is a stratified Datalog program. The answer to a Datalog query $q = (g, P)$ over a database D is $M[g]$, where M is the unique minimal model in $\mathcal{MM}(P \cup D)$, and will be denoted as $q(D)$.

Given a database D , a set of prioritized integrity constraints IC and a query $q = (g, P)$, the *preferred consistent answer* of the query q on the database D , denoted as $q(D, IC)$, gives three sets, denoted as $q(D, IC)^+$, $q(D, IC)^-$ and $q(D, IC)^u$. These contain, respectively, the sets of g -tuples which are *true* (i.e. belonging to $\bigcap_{R \in \text{repairs}_p(D, IC)} q(R(D))$), *false* (i.e. not belonging to $\bigcup_{R \in \text{repairs}_p(D, IC)} q(R(D))$) and *undefined* (i.e. set of tuples which are neither true nor false). It is worth noting that the preferred consistent answer of a query considers only the preferred repairs rather than all the repairs.

Desirable properties.

We now introduce desirable properties on the set of preferred repairs. Properties which should be satisfied by families of preferred repairs have been introduced in [28]. Here we adapt properties defined in [28], where preferences are static, to our framework.

Given a database D and a set of PAICs IC , then the following properties can be identified:

- **Non-emptiness:** $\langle D, IC \rangle$ always admits some preferred repairs.
- **Monotonicity:** adding preference information can only narrow the set of preferred repairs.
- **Non-discrimination:** if no preference information is expressed, any repair is a preferred repair.

Given two sets of PAICs IC_1 and IC_2 , we say that $IC_1 \triangleleft IC_2$ if IC_1 is derived from IC_2 by replacing one or more \succ symbols with the \vee symbol.

Definition 3.10. Given a set of PAICs IC and two relation symbols a and b , we say that $\pm a$ (inserting into or deleting from a) is preferable to $\pm b$ (inserting into or deleting from b) w.r.t. IC and we write $\pm a \gg_{IC} \pm b$, if:

1. there is a PAIC $c \in IC$ such that $Head(c) = \dots \pm a(X) \dots \succ \dots \pm b(Y) \dots$,
or
2. there exists $\pm c$ such that $\pm a \gg_{IC} \pm c$ and $\pm c \gg_{IC} \pm b$.

Observe that the above condition could be relaxed by considering ground atoms instead of predicate symbols.

The following theorem shows some properties of PAICs.

Theorem 3.3. Let D be a database and IC a set of PAICs.

1. **Non-emptiness:** $repairs_p(D, IC) \neq \emptyset$ if IC is satisfiable.
2. **Monotonicity:** $IC' \triangleleft IC \Rightarrow repairs_p(D, IC) \subseteq repairs_p(D, IC')$ if \gg_{IC} is acyclic, i.e. there does not exist an update atom $\pm a$ such that $\pm a \gg_{IC} \pm a$.¹
3. **Non-discrimination:**
if $IC = \mathcal{CC}(IC)$ then $repairs_p(D, IC) = repairs(D, IC)$.

3.3 Computing Repairs Through Datalog Programs

A general approach for the computation of repairs and consistent answers in the presence of databases with universal integrity constraints has been proposed in [49]. The technique is based on the generation of a disjunctive program $\mathcal{DP}(IC)$ derived from the set of integrity constraints IC . The repairs for a database D can be generated from the stable models of $\mathcal{DP}(IC) \cup D$. We now present an extension of this technique for prioritized active integrity constraints in canonical form.

Definition 3.11. Given a set of prioritized active integrity constraints IC , then $\mathcal{DP}(IC)$ is the disjunctive datalog program derived from IC (or equivalently from $\mathcal{SC}(IC)$) by replacing a PAIC of the form (3.2) with a disjunctive rule of the form:

$$\bigvee_{j=1}^m -b_j(X_j) \vee \bigvee_{j=m+1}^n +b_j(X_j) \leftarrow \bigwedge_{j=1}^m (b_j(X_j) \vee +b_j(X_j)), \\ \bigwedge_{j=m+1}^n (\text{not } b_j(X_j) \vee -b_j(X_j)), \varphi(X_0)$$

and by adding a constraint

$$\leftarrow -b(X), +b(X)$$

for each predicate symbol b .

¹ Note that also the approach proposed in [28] satisfies monotonicity property under the assumption that the priority relation is acyclic.

Example 3.5. Consider the set of integrity constraints IC of Example 3.3. The following set of rules $\mathcal{DP}(IC)$ can be derived:

$$\begin{aligned}
 +a \vee +b \vee -c &\leftarrow (c \vee +c), (not\ a \vee -a), (not\ b \vee -b) \\
 -c \vee +d &\leftarrow (c \vee +c), (not\ d \vee -d) \\
 &\leftarrow +a, -a \\
 &\leftarrow +b, -b \\
 &\leftarrow +c, -c \\
 &\leftarrow +d, -d
 \end{aligned}$$

It is worth noting that, in considering atoms of the form $+a(t)$, $-a(t)$ and $a(t)$, the symbols $+a$, $-a$ and a are assumed to be different predicate symbols.

Definition 3.12. Given an interpretation M , we denote as $UpdateAtoms(M)$ the set of update atoms in M . Given a set S of interpretations, we define $UpdateAtoms(S) = \{UpdateAtoms(M) \mid M \in S\}$.

Theorem 3.4. Given a database D and a set of canonical prioritized active integrity constraints IC , then:

- (Soundness) for every stable model M of $\mathcal{DP}(IC) \cup D$, $UpdateAtoms(M)$ is a repair for $\langle D, IC \rangle$;
- (Completeness) for every database repair S for $\langle D, IC \rangle$ there exists a stable model M of $\mathcal{DP}(IC) \cup D$ such that $S = UpdateAtoms(M)$. \square

Example 3.6. Consider the database $D = \{c\}$ and the set of PAICs IC of Example 3.3. The stable models of $\mathcal{DP}(IC) \cup D$ are $M_1 = \{c, -c\}$, $M_2 = \{c, +a, +d\}$ and $M_3 = \{c, +b, +d\}$. Each stable model corresponds to a repair for $\langle D, IC \rangle$ and vice versa.

Definition 3.13. Given a set of prioritized active integrity constraint IC , then $\mathcal{FP}(IC)$ is the datalog program obtained as follows:

- for each update atom $\pm a(X)$ defined in $\mathcal{DP}(IC)$, the following rule is introduced:

$$\leftarrow \pm a(X), not\ founded\ \pm a(X)$$

- if $\pm a(X)$ appears in the head of a constraint $c \in \mathcal{AC}(IC)$, that is c is of the form:

$$(\forall X)[\bigwedge_{j=1}^m b_j(X_j), \bigwedge_{j=m+1}^n not\ b_j(X_j), Comp(\pm a(X)), \varphi(X_0) \rightarrow \bigvee_{i=1}^p \pm a_i(Y_i) \vee \pm a(X)]$$

then the following rule is introduced:

$$\begin{aligned}
 founded\ \pm a(X) &\leftarrow Comp(\pm a(X)), \bigwedge_{j=1}^m ((b_j(X_j) \wedge not\ -b_j(X_j)) \vee +b_j(X_j)), \\
 &\quad \bigwedge_{j=m+1}^n (not\ b_j(X_j) \wedge not\ +b_j(X_j)) \vee -b_j(X_j), \varphi(X_0)
 \end{aligned}$$

Given a set of prioritized active integrity constraint IC , then $\mathcal{FDP}(IC) = \mathcal{DP}(IC) \cup \mathcal{FP}(IC)$.

Example 3.7. Consider the set of integrity constraints IC of Example 3.3. The following set of rules $\mathcal{FP}(IC)$ can be derived:

$$\begin{aligned} &\leftarrow +a, \text{ not } \text{founded}+a \\ &\leftarrow +b, \text{ not } \text{founded}+b \\ &\leftarrow -c, \text{ not } \text{founded}-c \\ &\leftarrow +d, \text{ not } \text{founded}+d \end{aligned}$$

$$\begin{aligned} \text{founded}+a &\leftarrow \text{not } a, ((c \wedge \text{not } -c) \vee +c), ((\text{not } b \wedge \text{not } +b) \vee -b) \\ \text{founded}+b &\leftarrow \text{not } b, ((c \wedge \text{not } -c) \vee +c), ((\text{not } a \wedge \text{not } +a) \vee -a) \\ \text{founded}-c &\leftarrow c, ((\text{not } b \wedge \text{not } +b) \vee -b), ((\text{not } a \wedge \text{not } +a) \vee -a) \\ \text{founded}-d &\leftarrow c, ((\text{not } d \wedge \text{not } +d) \vee -d) \end{aligned}$$

Theorem 3.5. Given a database D and a set of canonical prioritized active integrity constraints IC , then:

- (Soundness) for every stable model M of $\mathcal{FDP}(IC) \cup D$, $\text{UpdateAtoms}(M)$ is a founded repair for $\langle D, IC \rangle$;
- (Completeness) for every founded repair S for $\langle D, IC \rangle$ there exists a stable model M of $\mathcal{FDP}(IC) \cup D$ such that $S = \text{UpdateAtoms}(M)$. \square

Example 3.8. Consider the database $D = \{c\}$ and the set of PAICs IC of Example 3.3. The unique stable model of $\mathcal{FDP}(IC) \cup D$ is $M = \{c, -c\}$ whose update atoms correspond to the unique founded repair for $\langle D, IC \rangle$.

Observe that given a set of PAICs IC and a database D , then $\mathcal{DP}(IC) \cup D$ gives all the possible repairs for $\langle D, IC \rangle$, whereas $\mathcal{DP}(IC) \cup \mathcal{FP}(IC) \cup D$ gives only the founded repairs as the constraints in $\mathcal{FP}(IC)$ discard every stable model of $\mathcal{DP}(IC) \cup D$ which does not correspond to a founded repair.

Theorem 3.6. Let D be a database and IC a set of prioritized active integrity constraints, then

$$\text{repairs}_p(D, IC) \begin{cases} \subseteq \text{UpdateAtoms}(\text{SM}(\mathcal{FDP}(IC) \cup D)) & \text{if } \text{SM}(\mathcal{FDP}(IC) \cup D) \neq \emptyset \\ \subseteq \text{UpdateAtoms}(\text{SM}(\mathcal{DP}(IC) \cup D)) & \text{if } \text{SM}(\mathcal{FDP}(IC) \cup D) = \emptyset \end{cases}$$

\square

3.4 Discussion

We have introduced *prioritized active integrity constraints*, a simple and powerful form of active rules with declarative semantics, well suited for expressing preferences among repairs. A prioritized active integrity constraint defines an integrity constraint, the actions which should be performed if the constraint

is not satisfied and preferences among these actions. These preferences determine a partial order among feasible repairs, so that preferred repairs can be selected among all the possible repairs. It has been shown that prioritized active integrity constraints can be rewritten into disjunctive Datalog programs and that repairs can be computed through the computation of stable models.

The increased interest in preferences in logic programs is reflected by an extensive number of proposals and systems for preference handling. Most of the approaches propose an extension of Gelfond and Lifschitz's extended logic programming by adding preference information [32, 44, 70, 80]. The literature distinguish *static* and *dynamic* preferences. Static preferences are fixed at the time a theory is specified, i.e. they are "external" to the logic program [70], whereas dynamic preferences appear within the logic program and are determined "on the fly" [17, 32]. The most common form of preference consists in specifying a strict partial order on rules [32, 44, 70, 80], whereas more sophisticated forms of preferences also allow us to specify priorities between conjunctive (disjunctive) knowledge with preconditions [17, 70]. In [28] the framework of consistent query answer is extended by allowing preferences among tuples to be expressed. Several families of preferred repairs (i.e. subsets of repairs selected with priorities) have been also investigated.

Disjunctive Databases for Representing Repairs

In this chapter we address the problem of representing the set of repairs of a possibly inconsistent database by means of a disjunctive database. Specifically, the class of denial constraints is considered. We show that, given a database and a set of denial constraints, there exists a (unique) disjunctive database, called *canonical*, which represents the repairs of the database w.r.t. the constraints and is contained in any other disjunctive database with the same set of minimal models. We propose an algorithm for computing the canonical disjunctive database. Finally, we study the size of the canonical disjunctive database in the presence of functional dependencies for both repairs and cardinality-based repairs.

4.1 Introduction

Inconsistency leads to *uncertainty* as to the actual values of tuple attributes. Thus, it is natural to study the possible use of incomplete database frameworks in this context. The set of repairs for a possibly inconsistent database could be represented by means of an incomplete database whose possible worlds are exactly the repairs of the inconsistent database.

In this chapter, we consider a specific incomplete database framework: *disjunctive databases*. A disjunctive database is a finite set of disjunctions of facts. Its semantics is given by the set of minimal models. There is a clear intuitive connection between inconsistent and disjunctive databases. For instance, consider the following example.

Example 4.1. Consider the following relation r

<i>Name</i>	<i>Salary</i>	<i>Dept</i>
<i>john</i>	50	<i>cs</i>
<i>john</i>	100	<i>cs</i>

and the functional dependency $f : Name \rightarrow Salary Dept$ stating that each employee has a unique salary and a unique department. Clearly, r is inconsistent w.r.t. f as it stores two different salaries for the same employee $john$. Assuming that the database is viewed as a set of facts and the symmetric difference is used to capture the distance between two databases, there exist two repairs for r w.r.t. f , namely $\{employee(john, 50, cs)\}$ and $\{employee(john, 100, cs)\}$. These repairs could be represented by the disjunctive database $\mathcal{D} = \{employee(john, 50, cs) \vee employee(john, 100, cs)\}$, as the minimal models of \mathcal{D} are exactly the repairs of r w.r.t. f .

Disjunctive databases have been studied for a long time [55, 56, 66, 34]. More recently, they have again attracted attention in the database research community because of potential applications in data integration, extraction and cleaning [10]. Our approach should be distinguished from the approaches that rely on stable model semantics of *disjunctive logic programs with negation* to represent repairs of inconsistent databases [7, 20, 50].

In this chapter we address the problem of *representing* the set of repairs of a database w.r.t. a set of denial constraints by means of a disjunctive database (in other words, a disjunctive database whose minimal models are the repairs).

We show that, given a database and a set of denial constraints, there exists a unique, *canonical* disjunctive database which (a) represents the repairs of the database w.r.t. the constraints, and (b) is contained in any other disjunctive database having the same set of minimal models. We propose an algorithm for computing the canonical disjunctive database which in general can be of exponential size. Next, we study the size of the canonical disjunctive database in the presence of restricted functional dependencies. We show that the canonical disjunctive database is of linear size when only one key is considered, but it may be of exponential size in the presence of two keys or one non-key functional dependency. Finally, we demonstrate that these results hold also for a different, cardinality-based semantics of repairs [63].

The chapter is organized as follows. In Section 4.2, we introduce some basic notions in disjunctive databases. In Section 4.3, we present an algorithm to compute the canonical disjunctive database and show that this database is contained in any other disjunctive database with the same minimal models. In Section 4.4, we study the size of the canonical disjunctive databases in the presence of functional dependencies. In Section 4.5, we investigate the size of the canonical disjunctive databases under the cardinality-based semantics of repairs. Finally, in Section 4.6 we draw the conclusions and outline some possible future research topics.

4.2 Disjunctive Databases

A disjunctive database \mathcal{D} is a finite set of non-empty disjunctions of distinct facts. A disjunction containing exactly one fact is called a *singleton* disjunction. A set M of facts is a model of \mathcal{D} if $M \models \mathcal{D}$; M is minimal if there is no $M' \subset M$ s.t. $M' \models \mathcal{D}$. We denote by $\mathcal{MM}(\mathcal{D})$ the set of minimal models of \mathcal{D} . For a disjunction $d \in \mathcal{D}$, S_d denotes the set of facts appearing in d . Given two distinct disjunctions d_1 and d_2 in \mathcal{D} , we say that d_1 *subsumes* d_2 if the set of facts appearing in d_1 is a (proper) subset of the set of facts appearing in d_2 , i.e. $S_{d_1} \subset S_{d_2}$. Moreover, the reduction of \mathcal{D} , denoted by $reduction(\mathcal{D})$, is the disjunctive database obtained from \mathcal{D} by discarding all the subsumed disjunctions, that is

$$reduction(\mathcal{D}) = \{d \mid d \in \mathcal{D} \wedge \nexists d' \in \mathcal{D} \text{ s.t. } d' \text{ subsumes } d\}.$$

Observe that for any disjunctive database \mathcal{D} , $\mathcal{MM}(\mathcal{D}) = \mathcal{MM}(reduction(\mathcal{D}))$.

4.3 Representing Repairs Through Disjunctive Databases

In this section we propose an algorithm to compute a disjunctive database whose minimal models are the repairs of a given database w.r.t. a set of denial constraints. We show that the so computed disjunctive database is the canonical one, that is any other disjunctive database whose minimal models coincide with the repairs of the original database is a superset of the canonical one (containing, in addition, only disjunctions which are subsumed by disjunctions in the canonical disjunctive database).

We denote by $\mathcal{D}(D, F)$ the disjunctive database returned by Algorithm 1 with the input consisting of a database D and a set F of denial constraints. In the second step of the algorithm, every fact t s.t. $\{t\}$ is an edge of the conflict hypergraph is discarded.

The disjunctions introduced in the step 5 allow us to guarantee that the minimal models are maximal (consistent) subsets of D . Intuitively, a disjunction of the form $t \vee t_1 \vee \dots \vee t_n$ (which contains one fact from each edge containing t) prevents from having a model m of $\widehat{\mathcal{D}}$ which contains neither t nor the t_i 's as in this case m would not be maximal.

The disjunctions introduced in the step 9 allow us to guarantee that the minimal models of $\mathcal{D}(D, F)$ are consistent w.r.t. F . Specifically, the loop in lines 6–9 is performed until $\widehat{\mathcal{D}}$ satisfies the following property: for every edge $e = \{t_1, \dots, t_k\}$ of the conflict hypergraph ($k > 1$), if there are $t_1 \vee D_1, \dots, t_k \vee D_k \in \widehat{\mathcal{D}}$ s.t. each D_i is not an empty disjunction, then $\{D_1 \vee \dots \vee D_k\}$ is also in $\widehat{\mathcal{D}}$. As it is shown in the proof of Theorem 4.1, this property entails that every minimal model of $\widehat{\mathcal{D}}$ does not contain $\{t_1, \dots, t_k\}$. Observe that the loop ends when $\widehat{\mathcal{D}}$ does not change anymore; at each iteration new disjunctions are added

Algorithm 1**Input:** a database D and a set F of denial constraints**Output:** a disjunctive database whose minimal models are the repairs for D and F

```

1 :  $\widehat{\mathcal{D}} := \emptyset$ 
2 :  $D' := D - \{t \mid \{t\} \text{ is an edge of } \mathcal{G}_{D,F}\}$ 
3 : for each  $t \in D'$ 
4 :   Let  $edges_{D',F}(t) = \{e_1, \dots, e_n\}$ 
5 :    $\widehat{\mathcal{D}} := \widehat{\mathcal{D}} \cup \{t \vee t_1 \vee \dots \vee t_n \mid t_i \in e_i \text{ and } t_i \neq t \text{ for } i = 1..n\}$ 
6 : repeat until  $\widehat{\mathcal{D}}$  does not change
7 :   for each edge  $e = \{t_1, \dots, t_k\}$  in  $\mathcal{G}_{D',F}$ 
8 :     for each  $t_1 \vee D_1, \dots, t_k \vee D_k \in \widehat{\mathcal{D}}$  s.t.  $D_i$  is not an empty disjunction
       and  $D_i$  does not contain any fact  $t' \neq t_i$  in  $e$ ,  $i = 1..k$ 
9 :        $\widehat{\mathcal{D}} := \widehat{\mathcal{D}} \cup \{D_1 \vee \dots \vee D_k\}$ 
10 : return  $reduction(\widehat{\mathcal{D}})$ 

```

to $\widehat{\mathcal{D}}$. Since the number of disjunctions is bounded (if the original database has h facts, there cannot be more than $2^h - 1$ disjunctions) the algorithm always terminates. In the last step of the algorithm, subsumed disjunctions are deleted. The following theorem states the correctness of Algorithm 1.

Theorem 4.1. *Given a database D and a set F of denial constraints, the set of minimal models of $\mathcal{D}(D, F)$ coincides with the set of repairs of D w.r.t. F .*

Proof. Since the disjunctive database $\mathcal{D}(D, F)$ returned by Algorithm 1 is equal to $reduction(\widehat{\mathcal{D}})$ (step 10), then $\mathcal{M}\mathcal{M}(\mathcal{D}(D, F)) = \mathcal{M}\mathcal{M}(\widehat{\mathcal{D}})$. First we prove

(1) $rep(D, F) \subseteq \mathcal{M}\mathcal{M}(\widehat{\mathcal{D}})$ and next (2) $rep(D, F) \supseteq \mathcal{M}\mathcal{M}(\widehat{\mathcal{D}})$.

(1) Consider a repair r in $rep(D, F)$. First we show that (a) r is a model of $\widehat{\mathcal{D}}$ and next (b) that it is a minimal model.

(a) We prove that r satisfies each disjunction in $\widehat{\mathcal{D}}$ by induction. Specifically, as base case we consider the disjunctions introduced in the step 5 of the algorithm, whereas the inductive step refers to the disjunctions introduced in the step 9. Suppose by contradiction that r does not satisfy a disjunction $t \vee t_1 \vee \dots \vee t_n$ introduced in the step 5. Observe that $edges_{D',F}(t) \subseteq edges_{D,F}(t)$ and each edge e' in $edges_{D,F}(t) - edges_{D',F}(t)$ is s.t. there is a fact $t' \in e'$ s.t. $\{t'\}$ is an edge of $\mathcal{G}_{D,F}$ (clearly, $t' \notin r$). Since in each edge in $edges_{D,F}(t)$ there is a fact (different from t) which is not in r , then $r \cup \{t\}$ is consistent, which violates the maximality of r . The inductive step consists in showing that r satisfies any disjunction added to $\widehat{\mathcal{D}}$ in the step 9 assuming that r satisfies $\widehat{\mathcal{D}}$. A disjunction $D_1 \vee \dots \vee D_k$, where the D_i 's are not empty disjunctions, is added to $\widehat{\mathcal{D}}$ whenever there exist $t_1 \vee D_1, \dots, t_k \vee D_k$ in $\widehat{\mathcal{D}}$ s.t. $e = \{t_1, \dots, t_k\}$ is an edge of $\mathcal{G}_{D',F}$, and D_i does not contain any fact $t' \neq t_i$ in e , for $i = 1..k$. Since r satisfies all the disjunctions $t_1 \vee D_1, \dots, t_k \vee D_k$ and

does not contain some fact t_j in e (as e is an edge of $\mathcal{G}_{D,F}$ too), it satisfies the disjunction D_j and then $D_1 \vee \dots \vee D_k$ as well. Hence r is a model of $\widehat{\mathcal{D}}$. (b) We now show that r is a minimal model, reasoning by contradiction. Assume that there exists a model $m' \subset r$ and let t be a fact in r but not in m' . Observe that t is a conflicting fact (it cannot be the case that there is a model of $\widehat{\mathcal{D}}$ which does not contain a non-conflicting fact because the algorithm introduces, in the step 5, a singleton disjunction d for each non-conflicting fact d). Moreover, as r is a repair, t is s.t. $\{t\}$ is not an edge of $\mathcal{G}_{D,F}$ and then t is in D' . For each edge e_i in $edges_{D',F}(t) = \{e_1, \dots, e_n\}$ there is a fact $t_i \neq t$ which is not in r as it is consistent and $edges_{D',F}(t) \subseteq edges_{D,F}(t)$. The same holds for m' as it is a subset of r . Then, the disjunction $t \vee t_1 \vee \dots \vee t_n$ in $\widehat{\mathcal{D}}$ (added in the step 5) is not satisfied by m' , which contradicts that m' is a model. Hence r is a minimal model of $\widehat{\mathcal{D}}$.

(2) Consider a minimal model m in $\mathcal{MM}(\widehat{\mathcal{D}})$. We show first (a) that it is consistent w.r.t. F and then (b) that it is maximal.

(a) First of all, it is worth noting that $\widehat{\mathcal{D}}$ doesn't contain a singleton disjunction t s.t. t is a conflicting fact of D . This can be shown as follows. Two cases may occur: either $\{t\}$ is an edge of $\mathcal{G}_{D,F}$ or it is not. As for the first case, since we have proved above that each repair of D and F is a model of $\widehat{\mathcal{D}}$ and no repair contains t , it cannot be the case that t is a singleton disjunction of $\widehat{\mathcal{D}}$. Let us consider the second case. For any conflicting fact t in D s.t. $\{t\}$ is not an edge of $\mathcal{G}_{D,F}$, there exist a repair r_1 s.t. $t \in r_1$ and a repair r_2 s.t. $t \notin r_2$. As we have proved above, there are two minimal models of $\widehat{\mathcal{D}}$ corresponding to r_1 and r_2 , then it cannot be the case that $t \in \widehat{\mathcal{D}}$. We prove that m is consistent w.r.t. F by contradiction, assuming that m contains a set of facts t_1, \dots, t_k s.t. $e = \{t_1, \dots, t_k\}$ is in $\mathcal{G}_{D,F}$. Let $S_{t_i} = \{D \mid t_i \vee D \in \widehat{\mathcal{D}} \text{ and } D \neq \emptyset \text{ does not contain any fact } t' \neq t_i \text{ in } e\}$ for $i = 1..k$. Two cases may occur: either (a) there is a set S_{t_i} which is empty or (b) all the sets S_{t_i} are not empty. (a) Let t_j be a fact in e s.t. S_{t_j} is empty. It is easy to see that $m - \{t_j\}$ is a model, which contradicts the minimality of m . (b) For each $D_1 \in S_{t_1}, \dots, D_k \in S_{t_k}$, it holds that $D_1 \vee \dots \vee D_k \in \widehat{\mathcal{D}}$. Then there is a set S_{t_j} s.t. m satisfies each D in S_{t_j} , otherwise it would be the case that some $D_1 \vee \dots \vee D_k$ in $\widehat{\mathcal{D}}$, where D_i is in S_{t_i} for $i = 1..k$, is not satisfied. It is easy to see that $m - \{t_j\}$ is a model, which contradicts the minimality of m . Hence m is consistent w.r.t. F .

(b) Now we prove that m is a maximal (consistent) subset of D reasoning by contradiction, thus assuming that there exists $m' \supset m$ which is consistent. Let t be a fact in m' but not in m . Since m' is consistent, for each edge e_i in $edges_{D',F}(t) = \{e_1, \dots, e_n\}$ there is a fact $t_i \neq t$ which is not in m' . The same holds for m as it is a (proper) subset of m' . This implies that m doesn't satisfy the disjunction $t \vee t_1 \vee \dots \vee t_n$ in $\widehat{\mathcal{D}}$ (added in the step 5), thus contradicting

the fact the m is a model. Hence m is a maximal consistent subset of D , that is a repair. \square

Given a database D with n facts, a rough bound on the size of $\mathcal{D}(D, F)$ is that it cannot have more than $2^n - 1$ disjunctions and each disjunction contains at most n facts, for any set F of denial constraints (in the next section we will study more precisely the size of $\mathcal{D}(D, F)$ for special classes of denial constraints, namely functional dependencies and key constraints).

The following theorem allows us to identify all the disjunctive databases which have the same minimal models of a given disjunctive database. Specifically, it states that given a disjunctive database \mathcal{D} , any other disjunctive database with the same minimal models is a superset of $\text{reduction}(\mathcal{D})$ containing in addition only disjunctions subsumed by disjunctions in $\text{reduction}(\mathcal{D})$. This result allows us to state that there is a (unique) disjunctive database representing the repairs for a given database and a set of denial constraints which is contained in any other disjunctive database with the same set of minimal models. We call such a disjunctive database *canonical*. Algorithm 1 computes the canonical disjunctive database (see Corollary 4.1).

Theorem 4.2. *Given a disjunctive database \mathcal{D} , the set \mathcal{R} of all disjunctive databases having the same minimal models as \mathcal{D} is equal to:*

$$\mathcal{R} = \{ \mathcal{D}' \mid \text{reduction}(\mathcal{D}) \subseteq \mathcal{D}' \wedge \forall d' \in \mathcal{D}' - \text{reduction}(\mathcal{D}) \exists d \in \text{reduction}(\mathcal{D}) \text{ which subsumes } d' \}$$

Proof. We denote by $\mathcal{S}(\mathcal{D})$ the set of all the disjunctive databases whose minimal models are $\mathcal{MM}(\mathcal{D})$. In order to prove that $\mathcal{R} = \mathcal{S}(\mathcal{D})$, first we show that (1) each disjunctive database in \mathcal{R} is also in $\mathcal{S}(\mathcal{D})$ and next that (2) each disjunctive database in $\mathcal{S}(\mathcal{D})$ is in \mathcal{R} too.

(1) Consider a disjunctive database \mathcal{D}' in \mathcal{R} . It is easy to see that $\text{reduction}(\mathcal{D}') = \text{reduction}(\mathcal{D})$. As a disjunctive database and its reduction have the same minimal models, $\mathcal{MM}(\mathcal{D}') = \mathcal{MM}(\mathcal{D})$ and hence \mathcal{D}' is in $\mathcal{S}(\mathcal{D})$.

(2) We show that any disjunctive database not belonging to \mathcal{R} is not in $\mathcal{S}(\mathcal{D})$. We recall that for a disjunction d , S_d denotes the set of facts appearing in d . Consider a disjunctive database \mathcal{D}_{out} which is not in \mathcal{R} . Two cases may occur: (a) $\text{reduction}(\mathcal{D}) \not\subseteq \mathcal{D}_{out}$ or (b) $\text{reduction}(\mathcal{D}) \subseteq \mathcal{D}_{out}$ and $\exists d' \in \mathcal{D}_{out} - \text{reduction}(\mathcal{D})$ s.t. there is no $d \in \text{reduction}(\mathcal{D})$ which subsumes d' .

(a) As $\text{reduction}(\mathcal{D}) \not\subseteq \mathcal{D}_{out}$, there is a disjunction a in $\text{reduction}(\mathcal{D})$ which is not in \mathcal{D}_{out} . Two cases may occur:

- there exists $a_1 \in \mathcal{D}_{out}$ which subsumes a ;
- the previous condition does not hold.

Let us consider the first case and let I be the interpretation $S - S_{a_1}$ where S is the set of facts appearing in $\text{reduction}(\mathcal{D})$. It is easy to see that I is a model of $\text{reduction}(\mathcal{D})$ (the only disjunctions that I could not satisfy are those ones

that contain only facts in S_{a_1} ; such disjunctions are not in $reduction(\mathcal{D})$ as they subsume a and $reduction(\mathcal{D})$ does not contain two disjunctions s.t. one subsumes the other). Then, there exists $M \subseteq I$ which is a minimal model of $reduction(\mathcal{D})$. As $a_1 \in \mathcal{D}_{out}$, each model of \mathcal{D}_{out} contains a fact in S_{a_1} , then M is not a minimal model of \mathcal{D}_{out} and so $\mathcal{MM}(reduction(\mathcal{D})) \neq \mathcal{MM}(\mathcal{D}_{out})$. Hence $\mathcal{D}_{out} \notin \mathcal{S}(\mathcal{D})$.

We consider now the second case. We show that $\mathcal{D}_{out} \notin \mathcal{S}(\mathcal{D})$ in a similar way to the previous case. Let I be the interpretation $S - S_a$ where S is the set of facts appearing in \mathcal{D}_{out} . It is easy to see that I is a model of \mathcal{D}_{out} (the only disjunctions that I could not satisfy are those ones which contain only facts in S_a ; such disjunctions are not in \mathcal{D}_{out} as \mathcal{D}_{out} contains neither a nor a disjunction which subsumes a). Then, there exists $M \subseteq I$ which is a minimal model of \mathcal{D}_{out} . As $a \in reduction(\mathcal{D})$, each model of $reduction(\mathcal{D})$ contains a fact in S_a , then M is not a minimal model of $reduction(\mathcal{D})$; hence $\mathcal{D}_{out} \notin \mathcal{S}(\mathcal{D})$.

(b) Let I be the interpretation $S - S_{d'}$ where S is the set of facts appearing in $reduction(\mathcal{D})$. It is easy to see that I is a model of $reduction(\mathcal{D})$ (the only disjunctions that I could not satisfy are d' and those ones which subsume d'). Then, there exists $M \subseteq I$ which is a minimal model of $reduction(\mathcal{D})$. As $d' \in \mathcal{D}_{out}$, each model of \mathcal{D}_{out} contains a fact in $S_{d'}$, then M is not a minimal model of \mathcal{D}_{out} ; hence $\mathcal{D}_{out} \notin \mathcal{S}(\mathcal{D})$. \square

Corollary 4.1. *Given a database D and a set F of denial constraints, then $\mathcal{D}(D, F)$ is the canonical disjunctive database whose minimal models are the repairs for D and F .*

Proof. Straightforward from Theorem 4.1 and 4.2. \square

From now on, we will denote by $\mathcal{D}_{min}(D, F)$ the canonical disjunctive database whose minimal models are the repairs for a database D and a set F of denial constraints. Whenever D and F are clear from the context, we simply write \mathcal{D}_{min} instead of $\mathcal{D}_{min}(D, F)$.

4.4 Functional Dependencies

In this section we study the size of the canonical disjunctive database representing the repairs of a database in the presence of functional dependencies. Specifically, we show that when the constraints consist of only one key, the canonical disjunctive database is of linear size, whereas for one non-key functional dependency or two keys the size of the canonical database may be exponential.

We observe that in the presence of only one functional dependency, the conflict hypergraph has a regular structure that “induces” a regular disjunctive database which can be identified without performing Algorithm 1. When two

key constraints are considered, we are not able to provide such a characterization; this is because the conflict hypergraph can have an irregular structure and it is harder to identify a pattern for \mathcal{D}_{min} .

Given a disjunction d , we denote by $\|d\|$ the number of facts occurring in d . The size of a disjunctive database \mathcal{D} , denoted as $\|\mathcal{D}\|$, is the number of facts occurring in it, that is $\|\mathcal{D}\| = \sum_{d \in \mathcal{D}} \|d\|$. We study the size $\|\mathcal{D}_{min}\|$ of \mathcal{D}_{min} as a function of the size of the given database.

One key. Given a relation r and a key constraint k stating that the set X of attributes is a key of r , we denote by $cliques(r, k)$ the partition of r into $n = |\pi_X(r)|$ sets C_1, \dots, C_n , called *cliques*, s.t. each C_i does not contain two facts with different values on X . Observe that (i) facts in the same clique are pairwise conflicting with each other, (ii) the set of repairs of r w.r.t. k is $\{\{t_1, \dots, t_n\} \mid t_i \in C_i \text{ for } i = 1..n\}$.

Proposition 4.1. *Given a relation r and a key constraint k , then \mathcal{D}_{min} is equal to*

$$\{t_1 \vee \dots \vee t_m \mid \exists C = \{t_1, \dots, t_m\} \in cliques(r, k)\}$$

Proof. It is straightforward to see that the minimal models of the disjunctive database reported above are the repairs of r w.r.t. k ; since it coincides with its reduction, Theorem 4.2 implies that it is the canonical one. \square

It is easy to see that when one key constraint is considered, $\|\mathcal{D}_{min}\| = |r|$.

Proposition 4.2. *Given a relation and a key constraint, \mathcal{D}_{min} is computed in polynomial time by Algorithm 1.*

Proof. It is easy to see that after the first loop (steps 3-5) Algorithm 1 produces \mathcal{D}_{min} and, after that, step 9 is never performed. \square

Two keys. We now show that, in the presence of two key constraints, \mathcal{D}_{min} may have exponential size. Let D_n ($n > 0$) be the family of databases, containing $3n$ facts, of the following form:

	A	B
t_{11}	a	b_1
\vdots	\vdots	\vdots
t_{n1}	a	b_n
t_{12}	a_1	b_1
t_{13}	a_1	b'_1
\vdots	\vdots	\vdots
t_{n2}	a_n	b_n
t_{n3}	a_n	b'_n

Let $D \in D_n$ and A, B be two keys. The conflict hypergraph for D w.r.t. the two key constraints consists of the following edges:

$$\begin{aligned} & \{\{t_{i1}, t_{j1}\} \mid 1 \leq i, j \leq n \wedge i \neq j\} \cup \\ & \{\{t_{i1}, t_{i2}\} \mid 1 \leq i \leq n\} \cup \{\{t_{i2}, t_{i3}\} \mid 1 \leq i \leq n\} \end{aligned}$$

Thus, the conflict hypergraph contains a clique $\{t_{11}, \dots, t_{n1}\}$ of size n and, moreover, t_{i1} is connected to t_{i2} which is in turn connected to t_{i3} ($i = 1..n$).

Example 4.2. The conflict hypergraph for a database in D_4 , assuming that A and B are two keys, is reported in Figure 1.

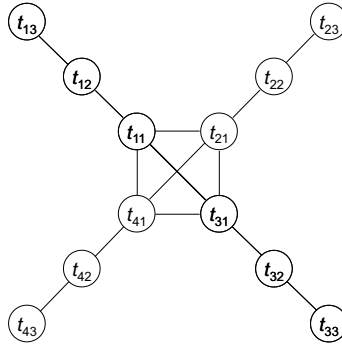


Fig. 4.1. Conflict hypergraph for a database in D_4 w.r.t. A, B key constraints

The following proposition identifies the canonical disjunctive database for a database in D_n for which A and B are keys; such a disjunctive database has exponential size.

Proposition 4.3. *Consider a database D in D_n and a set of constraints F consisting of two keys, A and B . Then \mathcal{D}_{min} is equal to \mathcal{D} where*

$$\begin{aligned} \mathcal{D} = & \{t_{i2} \vee t_{i3} \mid 1 \leq i \leq n\} \cup \\ & \{t_{i1} \vee t_{i2} \vee \bigvee_{z=1..n \wedge z \neq i} t_z \mid 1 \leq i \leq n \wedge t_z \in \{t_{z1}, t_{z3}\}\} \end{aligned}$$

Proof. First of all, we show that the minimal models of \mathcal{D} are the repairs of D w.r.t. F ; in particular we prove that (1) $\mathcal{MM}(\mathcal{D}) \subseteq \text{rep}(D, F)$ and (2) $\mathcal{MM}(\mathcal{D}) \supseteq \text{rep}(D, F)$.

(1) Consider a minimal model $m \in \mathcal{MM}(\mathcal{D})$. First we show that (a) m is consistent w.r.t. F and next (b) that it is maximal.

(a) Let E be the set of edges of $\mathcal{G}_{D, F}$. First we show that for each $e = \{t', t''\}$ in E and pair of disjunctions $d' = t' \vee D'$, $d'' = t'' \vee D''$ in \mathcal{D} s.t. D' (resp. D'') does not contain t'' (resp. t'), there is a disjunction in \mathcal{D} which is equal

to or subsumes $D' \vee D''$; next we show that this property implies that m is consistent w.r.t. F . We recall that E is the union of the following three sets:

$$E_1 = \{\{t_{i1}, t_{j1}\} \mid 1 \leq i, j \leq n \wedge i \neq j\}$$

$$E_2 = \{\{t_{i1}, t_{i2}\} \mid 1 \leq i \leq n\}$$

$$E_3 = \{\{t_{i2}, t_{i3}\} \mid 1 \leq i \leq n\}$$

Let us consider the case where $e \in E_1$, that is $e = \{t_{i1}, t_{j1}\}$ ($1 \leq i, j \leq n \wedge i \neq j$). Then a disjunction in \mathcal{D} containing t_{i1} but not t_{j1} is of the form

$$d'_1 : t_{i1} \vee t_{i2} \vee t_{j3} \vee \bigvee_{z=1..n \wedge z \neq i, j} t'_z$$

where $t'_z \in \{t_{z1}, t_{z3}\}$, or of the form

$$d'_2 : t_{h1} \vee t_{h2} \vee t_{i1} \vee t_{j3} \vee \bigvee_{z=1..n \wedge z \neq h, i, j} t'_z$$

where $1 \leq h \leq n \wedge h \neq i, j$ and $t'_z \in \{t_{z1}, t_{z3}\}$. Likewise, a disjunction in \mathcal{D} that contains t_{j1} but not t_{i1} is of the form

$$d''_1 : t_{j1} \vee t_{j2} \vee t_{i3} \vee \bigvee_{z=1..n \wedge z \neq i, j} t''_z$$

where $t''_z \in \{t_{z1}, t_{z3}\}$, or of the form

$$d''_2 : t_{k1} \vee t_{k2} \vee t_{j1} \vee t_{i3} \vee \bigvee_{z=1..n \wedge z \neq k, i, j} t''_z$$

where $1 \leq k \leq n \wedge k \neq i, j$ and $t''_z \in \{t_{z1}, t_{z3}\}$. In all the four possible cases, there is disjunction in \mathcal{D} which subsumes $D' \vee D''$:

- if $d' = d'_1$ and $d'' = d''_1$, then there exist both $t_{j2} \vee t_{j3}$ and $t_{i2} \vee t_{i3}$ in \mathcal{D} which subsume $D' \vee D''$;
- if $d' = d'_1$ and $d'' = d''_2$, then there exists $t_{i2} \vee t_{i3}$ in \mathcal{D} which subsumes $D' \vee D''$;
- if $d' = d'_2$ and $d'' = d''_1$, then there exists $t_{j2} \vee t_{j3}$ in \mathcal{D} which subsumes $D' \vee D''$;
- if $d' = d'_2$ and $d'' = d''_2$, then both $t_{h1} \vee t_{h2} \vee t_{i3} \vee t_{j3} \vee \bigvee_{z=1..n \wedge z \neq h, i, j} t'_z$ and $t_{k1} \vee t_{k2} \vee t_{i3} \vee t_{j3} \vee \bigvee_{z=1..n \wedge z \neq k, i, j} t''_z$, which are in \mathcal{D} , subsume $D' \vee D''$.

Let us consider the case where $e \in E_2$, namely $e = \{t_{i1}, t_{i2}\}$ ($1 \leq i \leq n$). A disjunction containing t_{i1} but not t_{i2} is of the form

$$t_{k1} \vee t_{k2} \vee t_{i1} \vee \bigvee_{z=1..n \wedge z \neq i, k} t_z$$

where $1 \leq k \leq n \wedge k \neq i$ and $t_z \in \{t_{z1}, t_{z3}\}$, whereas a disjunction containing t_{i2} but not t_{i1} is of the form $t_{i2} \vee t_{i3}$. Thus, $D' \vee D''$, which is equal to

$$t_{k1} \vee t_{k2} \vee t_{i3} \vee \bigvee_{z=1..n \wedge z \neq i, k} t_z$$

is in \mathcal{D} . Finally, consider the last case where $e \in E_3$, that is $e = \{t_{i2}, t_{i3}\}$ ($1 \leq i \leq n$). A disjunction containing t_{i2} but not t_{i3} is of the form

$$t_{i1} \vee t_{i2} \vee \bigvee_{z=1..n \wedge z \neq i} t'_z$$

where $t'_z \in \{t_{z1}, t_{z3}\}$, whereas a disjunction containing t_{i3} but not t_{i2} is of the form

$$t_{h1} \vee t_{h2} \vee t_{i3} \vee \bigvee_{z=1..n \wedge z \neq h, i} t''_z$$

where $1 \leq h \leq n \wedge h \neq i$ and $t''_z \in \{t_{z1}, t_{z3}\}$; $D' \vee D''$ is subsumed or equal to the disjunction

$$t_{h1} \vee t_{h2} \vee t_{i1} \vee \bigvee_{z=1..n \wedge z \neq h, i} t''_z$$

which is in \mathcal{D} .

Assume by contradiction that m is not consistent. Then there are two facts $t_a, t_b \in m$ s.t. $\{t_a, t_b\} \in E$. Let $S_{t_a} = \{D \mid t_a \vee D \in \mathcal{D} \text{ and } D \text{ does not contain } t_b\}$ and $S_{t_b} = \{D \mid t_b \vee D \in \mathcal{D} \text{ and } D \text{ does not contain } t_a\}$. As we have seen before, both these sets are not empty. We have previously proved that for each $D_a \in S_{t_a}$ and $D_b \in S_{t_b}$ there is a disjunction in \mathcal{D} which equals or subsumes $D_a \vee D_b$. Then, there is a set S_{t_x} among S_{t_a} and S_{t_b} s.t. m satisfies each D in S_{t_x} , otherwise there would be $D_a \in S_{t_a}, D_b \in S_{t_b}$ and a disjunction in \mathcal{D} which is equal to or subsumes $D_a \vee D_b$ which is not satisfied by m . Consider the interpretation $m' = m - \{t_x\}$ and let t_y be the fact among t_a and t_b which is not t_x . We now show that m' is a model, that contradicts the minimality of m . Clearly, m' satisfies every disjunction in \mathcal{D} which does not contain t_x . As for the disjunctions in \mathcal{D} containing t_x , it is easy to see that they are satisfied by m' : disjunctions containing t_y are satisfied since $t_y \in m'$, disjunctions not containing t_y are satisfied as well since m' satisfies every disjunction in S_{t_x} . Hence m is consistent w.r.t. F .

(b) Now we prove that m is a maximal (consistent) subset of D . First of all, we note that for each fact $t \in D$ there is a disjunction $t \vee t_1 \vee \dots \vee t_n$ in \mathcal{D} s.t. t_1, \dots, t_n are facts conflicting with t :

- for the facts t_{i2} and t_{i3} ($i = 1..n$) such disjunctions are $t_{i2} \vee t_{i3}$;
- for the facts t_{i1} ($i = 1..n$) such disjunctions are $t_{i1} \vee t_{i2} \vee \bigvee_{z=1..n \wedge z \neq i} t_{z1}$.

Assume by contradiction that m is not a maximal (consistent) subset of D . Then there exists $m' \supset m$ which is consistent. Let t be a fact in m' but not in m . Since m' is consistent, each fact conflicting with t is not in m' and, thus,

neither in m . This implies that m doesn't satisfy the disjunction $t \vee t_1 \vee \dots \vee t_n$ containing t and some fact conflicting with it: the fact that m is a model is contradicted.

(2) Consider a repair r for D and F . We show first (a) that r is a model of \mathcal{D} and next (b) that it is a minimal model.

(a) Suppose by contradiction that r is not a model of \mathcal{D} , then there is a disjunction $d \in \mathcal{D}$ which is not satisfied by r . Specifically, d is either of the form $t_{i2} \vee t_{i3}$ ($1 \leq i \leq n$) or $t_{i1} \vee t_{i2} \vee \bigvee_{z=1..n \wedge z \neq i} t_z$, $1 \leq i \leq n$ and $t_z \in \{t_{z1}, t_{z3}\}$. In the former case, $r \cup \{t_{i3}\}$ is consistent, since the only fact conflicting with t_{i3} , namely t_{i2} , is not in r . This contradicts the maximality of r . As for the latter case, let $T_3 = \{t_{j3} \mid t_{j3} \text{ appears in } d\}$. For each $t_{j3} \in T_3$ we have that $t_{j2} \in r$, because r does not contain t_{j3} and t_{j3} is conflicting only with t_{j2} (if t_{j2} was not in r , then r would not be maximal). Then for each $t_{j3} \in T_3$, since r contains t_{j2} , it does not contain t_{j1} otherwise it would not be consistent. Thus r does not contain any fact t_{k1} with $1 \leq k \leq n \wedge k \neq i$. Since r contains neither the facts t_{k1} 's nor t_{i2} , which are all the facts conflicting with t_{i1} , then $r \cup \{t_{i1}\}$ is consistent (observe that $t_{i1} \notin r$). This contradicts the maximality of r . Hence r is a model of \mathcal{D} .

(b) We now show that r is a minimal model of \mathcal{D} reasoning by contradiction. Assume that there exists a model $m' \subset r$ of \mathcal{D} and let t be a fact in r but not in m' . All the facts conflicting with t are not in r as r is consistent. The same holds for m' since it is a (proper) subset of r . We recall that for each fact $t' \in D$ there is a disjunction in \mathcal{D} containing t' and only facts conflicting with t' ; then there is a disjunction $d : t \vee t_1 \vee \dots \vee t_n$ in \mathcal{D} s.t. t_1, \dots, t_n are facts conflicting with t . Since m' does not satisfy d , it is not a model, thus we get a contradiction. Hence r is a minimal model of \mathcal{D} .

We have shown that the minimal models of \mathcal{D} are the repairs of D w.r.t. F . Since $\mathcal{D} = \text{reduction}(\mathcal{D})$, from Theorem 4.2 we have that \mathcal{D} is the canonical disjunctive database whose minimal models are the repairs of D w.r.t. F . \square

Corollary 4.2. *Consider a database D in D_n and let A and B be two keys; $|\mathcal{D}_{\min}| = 2n + (n + 1) \cdot n2^{n-1}$.*

Proof. From Proposition 4.3, it is easy to see that \mathcal{D}_{\min} contains n disjunctions of 2 facts and $n2^{n-1}$ disjunctions of $n + 1$ facts. \square

One functional dependency. Given a relation r and a functional dependency $f : X \rightarrow Y$, we denote by $\text{cliques}(r, f)$ the partition of r into $n = |\pi_X(r)|$ sets C_1, \dots, C_n , called *cliques*, s.t. each C_i does not contain two facts with different values on X . For each clique C_i in $\text{cliques}(r, f)$ we denote by $\text{clusters}(C_i)$ the partition of C_i into $m_i = |\pi_Y(C_i)|$ sets G_1, \dots, G_{m_i} , called *clusters*, s.t. each cluster doesn't contain two facts with different values on Y . It is worth noting that (i) facts in the same cluster are not conflicting each other, (ii) given two different clusters G_1, G_2 of the same clique, each

fact in G_1 (resp. G_2) is conflicting with every fact in G_2 (resp. G_1), (iii) the set of repairs of r w.r.t. f is $\{G_1 \cup \dots \cup G_n \mid G_i \in \text{clusters}(C_i) \text{ for } i = 1..n\}$.

Proposition 4.4. *Given a relation r and a functional dependency f , then \mathcal{D}_{min} is equal to \mathcal{D} where*

$$\mathcal{D} = \{t_1 \vee \dots \vee t_k \mid \exists C \in \text{cliques}(r, f) \text{ s.t. } \text{clusters}(C) = \{G_1, \dots, G_k\} \\ \text{and } t_1 \in G_1, \dots, t_k \in G_k\}$$

Proof. We show first (1) that each minimal model of \mathcal{D} is a repair for r and f and next (2) that each repair of r w.r.t. f is a minimal model of \mathcal{D} .

(1) Consider a minimal model m of \mathcal{D} . Let $\text{cliques}(r, f) = \{C_1, \dots, C_n\}$ be the cliques for r and f . For each clique C_i in $\text{cliques}(r, f)$ there is a cluster G_j in $\text{clusters}(C_i) = \{G_1, \dots, G_k\}$ s.t. $G_j \subseteq m$ (otherwise m would not satisfy the disjunction $t_1 \vee \dots \vee t_k$ in \mathcal{D} where $t_h \in G_h$ and $t_h \notin m$, $h = 1..k$). Let $\overline{G}_1, \dots, \overline{G}_n$ be such clusters, where each \overline{G}_l is a cluster of C_l for $l = 1..n$. Since $\overline{G}_1 \cup \dots \cup \overline{G}_n \subseteq m$ and $\overline{G}_1 \cup \dots \cup \overline{G}_n \models \mathcal{D}$, then $m = \overline{G}_1 \cup \dots \cup \overline{G}_n$, which is, as we have observed before, a repair.

(2) Consider a repair s in $\text{rep}(r, f)$. As s consists of one cluster for each clique, it is easy to see that s is a model of \mathcal{D} . We show that s is minimal by contradiction assuming that there exists $s' \subset s$ which is a model of \mathcal{D} . Let t be a fact in s which is not in s' . Let C_t and G_t be the clique and the cluster, respectively, containing t ; moreover let $\text{clusters}(C_t) = \{G_t, G_1, \dots, G_k\}$. The disjunction $t \vee t_1 \vee \dots \vee t_k$, where $t_i \in G_i$, $i = 1..k$, which is in \mathcal{D} , is not satisfied by s' as s' contains exactly one cluster per clique (thus it does not contain any fact in G_i , $i = 1..k$) and does not contain t . This contradicts the fact that s' is a model. So s is a minimal model of \mathcal{D} .

Hence the minimal models of \mathcal{D} are exactly the repairs for r and f ; as \mathcal{D} is equal to its reduction, Theorem 4.2 entails that $\mathcal{D} = \mathcal{D}_{min}$. \square

Clearly, the size of \mathcal{D}_{min} may be exponential if the functional dependency is a non-key dependency, as shown in the following example.

Example 4.3. Consider the relation r , consisting of $2n$ facts, reported below and the non-key functional dependency $A \rightarrow B$.

	A	B	C
t'_1	a	b_1	c_1
t''_1	a	b_1	c_2
\vdots	\vdots	\vdots	\vdots
t'_n	a	b_n	c_1
t''_n	a	b_n	c_2

There is a unique clique consisting of n clusters $G_i = \{t'_i, t''_i\}$, $i = 1..n$. Then $\mathcal{D}_{min} = \{t_1 \vee \dots \vee t_n \mid t_i \in G_i \text{ for } i = 1..n\}$ and $|\mathcal{D}_{min}| = n2^n$.

4.5 Cardinality-based Repairs

In this section we consider *cardinality-based repairs*, that is consistent databases which minimally differ from the original database in terms of the number of facts in the symmetric difference (in the previous sections we have considered consistent databases for which the symmetric difference is minimal under set inclusion, we will refer to them as *S-repairs*).

We show that, likewise to what has been presented in Section 4.4, the size of the canonical disjunctive database (representing the cardinality-based repairs) is linear when only one key constraint is considered, whereas it may be exponential when two keys or one non-key functional dependency are considered.

It is easy to see that in the presence of only one key constraint the cardinality-based repairs coincide with the S-repairs, so the canonical disjunctive database is of linear size.

When the constraints consists of one functional dependency, it is easy to see that if for every clique its clusters have the same cardinality, then the cardinality-based repairs coincide with the S-repairs. This is the case for the database of Example 4.3, where the size of the canonical disjunctive database is exponential.

Finally, we consider the case where two key constraints are considered. We directly show that the size of the canonical disjunctive database is also exponential.

Lemma 4.1. *Consider a database D in D_n and a set of integrity constraints F consisting of two keys, A and B . Then the set of S-repairs is equal to R where*

$$R = \{\{t_{12}, \dots, t_{n2}\}\} \cup \{\{t_{i1}, t_{i3}\} \cup \bigcup_{j=1..n \wedge j \neq i} \{t_j\} \mid 1 \leq i \leq n \wedge t_j \in \{t_{j2}, t_{j3}\}\}$$

Proof. It is easy to see that each database in R is a S-repair.

Consider a S-repair r of D w.r.t. F . We show that r is in R using reasoning by cases:

- Suppose that $t_{13} \in r$. Then $t_{12} \notin r$ and either (1) $t_{11} \in r$ or (2) $t_{11} \notin r$.
 1. Since $t_{11} \in r$, for $j = 2..n$ $t_{j1} \notin r$ and either t_{j2} or t_{j3} is in r , that is $r = \{t_{11}, t_{13}, t_2, \dots, t_n\}$ where $t_j \in \{t_{j2}, t_{j3}\}$, $j = 2..n$. It is easy to see that $r \in R$.
 2. Since $t_{11} \notin r$, there exists $t_{k1} \in r$ with $2 \leq k \leq n$. Then $t_{k2} \notin r$ and $t_{k3} \in r$. For $j = 2..n \wedge j \neq k$, $t_{j1} \notin r$ and either t_{j2} or t_{j3} is in r , that is $r = \{t_{13}, t_{k1}, t_{k3}\} \cup \bigcup_{j=2..n \wedge j \neq k} \{t_j\}$ where $t_j \in \{t_{j2}, t_{j3}\}$. Clearly, $r \in R$.
- Suppose that $t_{13} \notin r$. Then $t_{12} \in r$ and $t_{11} \notin r$. Two cases may occur: either (1) there exists $t_{k1} \in r$ with $2 \leq k \leq n$ or (2) $t_{j1} \notin r$ for $j = 1..n$.

1. Since $t_{k1} \in r$ then $t_{k2} \notin r$ and $t_{k3} \in r$. For $j = 2..n \wedge j \neq k$ $t_{j1} \notin r$ and either t_{j2} or t_{j3} is in r , that is $r = \{t_{12}, t_{k1}, t_{k3}\} \cup \bigcup_{j=2..n \wedge j \neq k} \{t_j\}$ where $t_j \in \{t_{j2}, t_{j3}\}$. It is easy to see that $r \in R$.
2. $r = \{t_{12}, \dots, t_{n2}\}$ which is in R . □

Corollary 4.3. *Consider a database D in D_n and a set of integrity constraints F consisting of two keys, A and B . Then the set of cardinality-based repairs is*

$$\{ \{t_{i1}, t_{i3}\} \cup \bigcup_{j=1..n \wedge j \neq i} \{t_j\} \mid 1 \leq i \leq n \wedge t_j \in \{t_{j2}, t_{j3}\} \}$$

Proof. Straightforward from Lemma 4.1. □

The following proposition identifies the canonical disjunctive database for a database in D_n for which A and B are keys; such a disjunctive database is of exponential size. In the following proposition and corollary, \mathcal{D}_{min} denotes the canonical disjunctive database representing the set of cardinality-based repairs.

Proposition 4.5. *Consider a database D in D_n and a set of integrity constraints F consisting of two keys, A and B . Then the canonical disjunctive database \mathcal{D}_{min} is equal to \mathcal{D} where*

$$\mathcal{D} = \{t_{i2} \vee t_{i3} \mid 1 \leq i \leq n\} \cup \{t_1 \vee \dots \vee t_n \mid t_i \in \{t_{i1}, t_{i3}\}, i = 1..n\}$$

Proof. We first show that (1) each cardinality-based repair of D w.r.t. F is a minimal model of \mathcal{D} and next that (2) each minimal model of \mathcal{D} is a cardinality-based repair.

(1) Consider a cardinality-based repair r of D w.r.t. F . We show first that (a) r is a model of \mathcal{D} and next that (b) it is a minimal model.

(a) From Corollary 4.3, it is easy to see that r satisfies each disjunction $t_{i2} \vee t_{i3}$ in \mathcal{D} , $1 \leq i \leq n$. Since Corollary 4.3 entails that there exists $1 \leq j \leq n$ s.t. $\{t_{j1}, t_{j3}\} \subseteq r$, then r satisfies each disjunction $t_1 \vee \dots \vee t_n$ in \mathcal{D} (where $t_i \in \{t_{i1}, t_{i3}\}$, $i = 1..n$). Thus r is a model of \mathcal{D} .

(b) We observe that for each fact $t \in D$ there is a disjunction $t \vee t_1 \vee \dots \vee t_n$ in \mathcal{D} s.t. t_1, \dots, t_n are facts conflicting with t : for the facts t_{i2} and t_{i3} ($i = 1..n$) such disjunctions are $t_{i2} \vee t_{i3}$; for the facts t_{i1} ($i = 1..n$) there is the disjunction $t_{i1} \vee \dots \vee t_{n1}$. In the same way as in Proposition 4.3, it can be shown that r is a minimal model of \mathcal{D} .

(2) Consider a minimal model m of \mathcal{D} . The fact that m is a S-repair of D w.r.t. F can be shown in the same way as in Proposition 4.3.

It is easy to see that $\{t_{12}, \dots, t_{n2}\}$ is not a model of \mathcal{D} and then, from Lemma 4.1 and Corollary 4.3, m is a cardinality-based repair of D w.r.t. F .

We have shown that \mathcal{D} represents the cardinality-based repairs of D w.r.t. F ; since $\mathcal{D} = \text{reduction}(\mathcal{D})$, from Theorem 4.2 we have that \mathcal{D} is the canonical one. \square

Corollary 4.4. *Consider a database D in D_n and let A and B be two keys; $\|\mathcal{D}_{min}\| = 2n + n2^n$.*

Proof. From Proposition 4.5, it is easy to see that \mathcal{D}_{min} contains n disjunctions of 2 facts and 2^n disjunctions of n facts. \square

4.6 Discussion

In this chapter we have addressed the problem of representing, by means of a disjunctive database, the set of repairs of a database w.r.t. a set of denial constraints. We have shown that, given a database and a set of denial constraints, there exists a unique canonical disjunctive database representing their repairs: any disjunctive database with the same set of minimal models is a superset of the canonical one, containing in addition disjunctions which are subsumed by the disjunctions in the canonical one. We have proposed an algorithm to compute the canonical disjunctive database. We have shown that the size of the canonical disjunctive database is linear when only one key is considered, but it may be exponential in the presence of two keys or one non-key functional dependency. We have shown that these results hold also when cardinality-based repairs are considered.

Future work in this area could explore different representations for the set of repairs. For instance, one can consider formulas with negation or non-clausal formulas. Such formulas can be more succinct than disjunctive databases, making query evaluation, however, potentially harder. We also observe that in the case of the repairs of a single relation the resulting disjunctive database consists of disjunctions of elements of this relation. It has been recognized that such disjunctions should be supported by database management systems [10]. Moreover, one could consider *restricting* inconsistent databases in such a way that the resulting repairs can be represented by relational databases with *OR-objects* [55]. In this case, one could use the techniques for computing *certain* query answers over databases with OR-objects [56] to compute *consistent* query answers over inconsistent databases. Finally, other kinds of representations of sets of possible worlds, e.g., *world-set decompositions* [3], should be considered. For example, the set of repairs of the database in Example 4.3 can be represented as a world-set decomposition of polynomial size.

Polynomial Time Queries over Inconsistent Databases

This chapter investigates the problem of repairing and querying relational databases which may be inconsistent with respect to functional dependencies and foreign key constraints. Specifically, particular sets of functional dependencies, called *canonical*, are considered.

We present a repairing strategy whereby only tuple updates and insertions are allowed in order to restore consistency: when foreign key constraints are violated, new tuples (possibly containing *null values*) are inserted into the database; when functional dependency violations occur, tuple updates (possibly introducing *unknown values*) are performed. We propose a semantics of constraint satisfaction for databases containing null and unknown values, since the repairing process can lead to such databases. The proposed approach allows us to obtain a unique repaired database which can be computed in polynomial time. The result of the repairing technique is an incomplete database (in particular, an OR-database). The *consistent* query answers over an inconsistent database are the *certain* answers on the repaired database. Relying on the results on the complexity of query processing in OR-databases, we identify conjunctive queries for which consistent answers can be computed in polynomial time.

5.1 Introduction

In this chapter we deal with the problem of repairing and querying databases in the presence of functional dependencies and foreign key constraints. Specifically, we consider particular sets of functional dependencies (called *canonical*) where attributes appearing on the right-hand side of functional dependencies cannot appear also on the left-hand side. We propose a repairing strategy which aims at preserving the information in the original database as much as possible: when foreign key constraints are violated new tuples are inserted into the database, whereas tuples updates are performed to make the database

consistent w.r.t. functional dependencies; thus tuple deletions are never performed. Since tuple insertions and updates may introduce respectively null and unknown values in the database, we propose a semantics of constraint satisfaction for databases containing null and unknown values. Let us give the basic idea of our approach in the following example.

Example 5.1. Consider the following database:

<i>Project</i>	
<i>Name</i>	<i>Manager</i>
<i>p1</i>	<i>john</i>
<i>p1</i>	<i>bob</i>
<i>p2</i>	<i>carl</i>

<i>Employee</i>	
<i>Name</i>	<i>Phone</i>
<i>john</i>	123
<i>bob</i>	111

Suppose to have the following set of constraints (functional dependencies and foreign key constraints):

- $fd_1 : Name \rightarrow Manager$ defined over *Project*,
- $fd_2 : Name \rightarrow Phone$ defined over *Employee*,
- $fk : Project[Manager] \subseteq Employee[Name]$.

The database is inconsistent as it violates both fd_1 and fk : there are two different managers for the same project *p1*, and the manager *carl*, appearing in the project relation, is not in the employee relation. In this case, the repaired (consistent) database is as follows:

<i>Project</i>	
<i>Name</i>	<i>Manager</i>
<i>p1</i>	#1
<i>p2</i>	<i>carl</i>

<i>Employee</i>	
<i>Name</i>	<i>Phone</i>
<i>john</i>	123
<i>bob</i>	111
<i>carl</i>	\perp_1

where #1 is an *unknown value* whose domain is $\{john, bob\}$ whereas \perp_1 is a (labeled) *null value*. Therefore, in order to satisfy the functional dependency fd_1 we have introduced the unknown value #1 which expresses the fact that *p1* has a unique manager that could be either *john* or *bob*. Observe that the first tuple in the project relation does not lead to a violation of fk because the manager of *p1*, whoever he may be, is in the employee relation too. The consistency of the original database w.r.t. fk has been restored by inserting the manager *carl* into the employee relation.

In the example above, observe that a labeled null value has been introduced for the phone number of *carl* since this information is missing. Specifically, we know neither if the telephone number of *carl* does not exist nor if the telephone number exists but is not known. Thus, neither the “nonexistent” (a value does not exist) nor the “unknown” (a value exists but it is not known) interpretation of the null is applicable in this situation. Here the null value

is interpreted as “no information” [78, 79], that is a placeholder for either a nonexistent or an unknown value. Thus, both unknown and null values express incomplete information, even though unknown values are “more informative than” null values.

As it will be shown in the chapter, given an inconsistent database and a set of constraints consisting of functional dependencies and foreign key constraints, the proposed repairing strategy allows us to obtain a unique repaired database which can be computed in polynomial time.

It is worth noting that, the so obtained (incomplete) database represents a set of “possible worlds”, namely the databases which are obtained by replacing every unknown value with a constant of its domain. The “certain answers” to a query over such a database are those tuples which can be derived from every possible world [56, 55, 54]. Observe that, in our case, a possible world can contain labeled nulls; the evaluation of a query over such a database treats each labeled null like a standard constant. We propose a semantics of query answering over inconsistent databases which naturally follows from the previous observations: the *consistent answers* to a query over a possibly inconsistent database are the certain answers in the repaired database.

Example 5.2. Consider the database of Example 5.1. The consistent answer to the query asking for the manager of $p2$ is *carl*, because this answer can be obtained from every possible world of the repaired database. Clearly, there is no consistent answer to the query asking for the manager of $p1$. Observe that, the consistent answer to the query asking for the telephone number of $p2$'s manager is \perp_1 , that means that we have no information about it.

Since repaired databases are OR-databases, on the complexity of query processing in OR-databases, we identify conjunctive queries for which consistent answers can be computed in polynomial time.

The rest of the chapter is organized as follows. Section 5.2 introduces a semantics of constraint satisfaction for databases containing null and unknown values, and a repairing strategy. The problem of querying inconsistent databases is tackled in Section 5.3. Finally, Section 5.4 contains concluding remarks and related work.

5.2 Repairing Inconsistent Databases

In this section we investigate the problem of repairing databases that are inconsistent w.r.t. functional dependencies and foreign key constraints. Specifically, in this chapter we consider particular sets of functional dependencies, called *canonical*, which are introduced in the following definition.¹

¹ In the rest of this chapter, we assume that functional dependencies are nontrivial and in standard form.

Definition 5.1. *Canonical sets of functional dependencies.* Let FD be a set of functional dependencies over the relation schema $r(U)$. FD is said to be in *canonical form* if for each functional dependency $X \rightarrow A \in FD$ does not exist a functional dependency $Y \rightarrow B \in FD$ such that $A \in Y$.

Thus, a set of functional dependencies is canonical if there is no attribute appearing in the right-hand side of a functional dependency and in the left-hand side of another one. When the database schema consists of more than one relation schema, each schema is associated with a canonical set of functional dependencies. It is easy to see that, given a canonical set FD of functional dependencies over a relation schema $r(U)$, there exists a unique key K of r and, moreover, every functional dependency $X \rightarrow A$ in FD is s.t. $X \subseteq K$. Thus, we observe that we deal with primary foreign key constraints.

Most of the proposed approaches for repairing inconsistent databases rely on tuple insertions and deletions; only a few works have investigated the computation of repairs by means of tuple updates ([76]). We propose a repairing strategy which aims at preserving the information in the original database as much as possible: when foreign key constraints are violated new tuples are inserted into the database whereas tuples updates are performed to make the database consistent w.r.t. functional dependencies; thus tuple deletions are never performed. As it has been shown in Example 5.1, null and unknown values can be introduced when repairing w.r.t. foreign key constraints and functional dependencies, respectively.

In the rest of this section, first we present a semantics of constraint satisfaction for databases containing null and unknown values, next we propose a repairing strategy.

5.2.1 Semantics of Constraint Satisfaction

First of all, let us introduce databases containing null and unknown values. We assume to have two infinite enumerable domains $D_{\#} = \{\#1, \dots, \#n, \dots\}$ and $D_{\perp} = \{\perp_1, \dots, \perp_k, \dots\}$ of distinct *unknown values* and distinct *labeled nulls*, respectively. The database domain, denoted as $Dom_{\#,\perp}$, contains, in addition to a set Dom of standard constants, the unknown values $D_{\#}$ and the null values D_{\perp} , i.e. $Dom_{\#,\perp} = Dom \cup D_{\#} \cup D_{\perp}$ (likewise, given an attribute A_i , its domain is $Dom_{\#,\perp}(A_i) = Dom(A_i) \cup D_{\#} \cup D_{\perp}$ where $Dom(A_i)$ is the set of constants for the attribute A_i). The sets of constants, null and unknown values are pairwise disjoint. A relation R over the schema $r(A_1, \dots, A_n)$ is a subset of $Dom_{\#,\perp}(A_1) \times \dots \times Dom_{\#,\perp}(A_n)$, where each unknown value $\#j$ appearing in R in correspondence of an attribute A_i is associated with a finite set of constants $dom(\#j) \subseteq Dom(A_i)$ (we call $dom(\#j)$ the domain of $\#j$). Given a tuple $t \in R$, we denote by $ground(t)$ the set of tuples obtained from t by replacing every unknown value occurring in t with a constant from its domain. We will call databases containing neither null nor unknown values *complete databases*. We point out that source databases are complete (and

possibly inconsistent) and the proposed repairing strategy leads to consistent databases possibly containing null and unknown values.

We present first a semantics of constraint satisfaction for databases containing null and unknown values in the presence of foreign key constraints, and next a semantics for functional dependencies (in this chapter we consider only canonical sets of functional dependencies, see Definition 5.1).

Definition 5.2. *Satisfaction of foreign key constraints.* Let R, S be two relations with schemata $r(U), s(V)$ respectively, and fk be a foreign key constraint of the form $r(X) \subseteq s(Y)$. R and S satisfy fk if for each tuple $t_R \in R$

- there exists X_i in X s.t. $t_R[X_i] = \perp_j$, or
- for each tuple $t'_R \in \text{ground}(t_R[X])$ there is a tuple $t_S \in S$ s.t. $t'_R = t_S[Y]$.

Clearly, a database D satisfies a set FK of foreign key constraints if it satisfies every foreign key constraint in FK .

In the above definition, if a tuple $t \in R$ contains a null value on some attribute in X , then it does not violate the foreign key constraint fk . Observe that, if such a tuple could violate fk , a reasonable way (relying on tuple insertions) to repair the database would lead to the insertion in S of a new tuple containing null values on attributes belonging to the primary key, which is not desirable (we recall that we deal with primary foreign key constraints). If a tuple $t \in R$ does not contain any null value on X , intuitively, it could be any tuple in $\text{ground}(t_R)$ and, in order to consider the constraint not violated by t_R , we require that any tuple in $\text{ground}(t_R)$ does not violate the constraint (under the standard semantics). When complete databases are considered, the previous definition coincides with the classical semantics of foreign key constraint satisfaction.

Example 5.3. Consider the database schema of Example 5.1 and the following database.

<i>Project</i>		<i>Employee</i>	
<i>Name</i>	<i>Manager</i>	<i>Name</i>	<i>Phone</i>
p1	\perp_1	john	123
p2	#1		

where $\text{dom}(\#1) = \{john, bob\}$. The first tuple of the project relation does not lead to a violation of $\text{Project}[\text{Manager}] \subseteq \text{Employee}[\text{Name}]$ since it has a null value on the attribute *Manager*. With regard to the second tuple in the project relation, since the manager of *p2* could possibly be *bob* and he is missing in the employee relation, the database is inconsistent.

Now we present the semantics of constraint satisfaction w.r.t. functional dependencies. As it has been observed before, given a canonical set FD of

functional dependencies over a relation schema $r(U)$, every functional dependency $X \rightarrow Y$ in FD is s.t. each attribute $X_i \in X$ belongs to the primary key of r . Since null values cannot occur in correspondence of such attributes, in the following definition we assume that, given a relation R over $r(U)$, every tuple $t \in R$ contains constants or unknown values on attributes appearing in the left-hand side of some functional dependency. Moreover, for the sake of simplicity of presentation, we say that $dom(c) = \{c\}$ for every constant $c \in Dom$.

Definition 5.3. *Satisfaction of functional dependencies.* Given a relation R and a functional dependency $fd = X \rightarrow A$ over the schema $r(U)$, R satisfies fd if for every pair t_1, t_2 of tuples in R , $\bigwedge_{X_i \in X} (dom(t_1[X_i]) \cap dom(t_2[X_i]) \neq \emptyset)$ implies $t_1[A] = t_2[A]$. Clearly, R satisfies a set FD of functional dependencies if it satisfies every functional dependency in FD .

In the previous definition, similarly to the semantics for foreign key constraints, if two tuples in R could possibly have the same value on X , thus we require that they have same value on A in order to consider fd not violated. In the presence of complete databases the above definition coincides with the classical semantics of satisfaction of functional dependencies.

Example 5.4. Consider the database schema of Example 5.1 and the following database (the employee relation is empty and then omitted).

<i>Project</i>	
<i>Name</i>	<i>Manager</i>
<i>p1</i>	\perp_1
<i>p1</i>	$\#1$
<i>p1</i>	<i>carl</i>

where $dom(\#1) = \{john, bob\}$. The tuples in the above relation are pairwise violating $fd_1 : Name \rightarrow Manager$ as they have the same information on *Name* but different information on *Manager*.

5.2.2 Repairing

We now present how to repair inconsistent databases. Basically, we introduce a rule to be applied whenever functional dependency violations occur, and another rule to be applied when foreign key constraints are violated. Our repairing strategy consists in applying these rules in some arbitrary order as long as they are applicable. We show that this procedure always terminates and the final consistent database does not depend on the order the rules have been applied. Finally, we show that the repairing process is polynomial time.

Let D be a database and FD, FK be sets of functional dependencies and foreign key constraints respectively. The aforementioned rules are the following:

- **FKC rule:** if there exist two relations R and S in D with schemata $r(U)$ and $s(V)$ respectively, a foreign key constraint $fk : r(X) \subseteq s(Y)$ in FK and a tuple $t_R \in R$ which violate fk (according to Definition 5.2), then we say that the *FKC rule is applicable*. The rule is applied as follows: for each tuple $t \in \text{ground}(t_R[X])$ s.t. there is no tuple $t_S \in S$ s.t. $t_S[Y] = t$ insert a tuple t_{new} into S s.t. $t_{new}[Y] = t$ and $\forall A_i \in (V - Y) t_{new}[A_i] = \perp_j$ where \perp_j is a fresh labeled null.
- **FDC rule:** if there exist a functional dependency $fd : X \rightarrow A \in FD$ and a relation R over $r(U)$, and two tuples t_1, t_2 in R which violate fd (according to Definition 5.3), then we say that the *FDC rule is applicable*. For the sake of simplicity of presentation, we say that $\text{dom}(\perp_i) = \emptyset$ for every unknown value $\perp_i \in D_\perp$ and $\text{dom}(c) = \{c\}$ for every constant $c \in \text{Dom}$. The rule is applied as follows. Let $d = \text{dom}(t_1[A]) \cup \text{dom}(t_2[A])$. If $d = \{c\}$ then $t_1[A] := c, t_2[A] := c$. If $d = \emptyset$ then $t_1[A] := \perp_j, t_2[A] := \perp_j$, where \perp_j is a fresh null value, and every occurrence of the old values $t_1[A]$ and $t_2[A]$ elsewhere is replaced with \perp_j . If both the previous cases do not hold then $t_1[A] := \#i, t_2[A] := \#i$, where $\#i$ is a fresh unknown value with domain $\text{dom}(\#i) = d$, and every occurrence of the old values $t_1[A]$ and $t_2[A]$ elsewhere, only if unknown values, is replaced with $\#i$.

Therefore, when a foreign key constraint is violated, the missing information is simply inserted into the database. By inserting a new tuple into the database, it may be the case that some information about the new tuple is missing and then we need to use null values. We adopt the no information interpretation of null values because, as it has been shown in Example 5.1, this interpretation allows us to model every kind of missing information, whereas adopting the unknown or the nonexistent interpretation non-factual information can be stored in the database.

Example 5.5. Consider the database of Example 5.3. By applying the FKC rule, the consistent database reported below is obtained.

<i>Project</i>		<i>Employee</i>	
<i>Name</i>	<i>Manager</i>	<i>Name</i>	<i>Phone</i>
p1	\perp_1	john	123
p2	#1	bob	\perp_2

Thus, we have inserted *bob* into the employee relation as it could be *p2*'s manager.

The proposed rule for repairing w.r.t. functional dependencies stems from the observation that, if a relation contains two tuples t_1, t_2 which are equal on a set X of attributes, and a functional dependency $X \rightarrow A$ is defined, this means that $t_1[X]$ (equivalently, $t_2[X]$) should be associated with a unique A -value. When this is not the case, that is t_1 and t_2 violate $X \rightarrow A$, we modify both of them on the attribute A in such a way that they have the

same information on this attribute. Specifically, when t_1, t_2 do not have a null value on A we “fix” them by assigning the same (unknown) value on A ; the possible values for this unknown value come from the old values of t_1 and t_2 on A . Observe that the domain of an unknown value consists of constants which come from the original (complete) database, and, in particular, these values were in the original database in positions where the unknown value occurs.

Example 5.6. Consider the database schema of Example 5.1 and the following database (the employee relation is empty and then omitted).

<i>Project</i>	
<i>Name</i>	<i>Manager</i>
p1	#1
p1	carl

where $dom(\#1) = \{john, bob\}$. Clearly, the database violates $fd_1 : Name \rightarrow Manager$. By applying the FDC rule, we obtain the following consistent database:

<i>Project</i>	
<i>Name</i>	<i>Manager</i>
p1	#2

where $dom(\#2) = \{john, bob, carl\}$.

Let us consider the case where $t_1[A]$ and $t_2[A]$ contain null values. If $t_1[A] = \perp_i$ and $t_2[A] = \perp_j$, with $i \neq j$, then the same (fresh) labeled null is assigned to both $t_1[A]$ and $t_2[A]$ (every occurrence of \perp_i and \perp_j elsewhere is replaced with the new labeled null). Let us now consider the case where there is only one null values among $t_1[A]$ and $t_2[A]$. As we have stressed several times, a null value is interpreted as no information (the value either exists but it is not known or does not exist) whereas we can say more about unknown values (a value exists but it is not know and a finite set of possible values is known). In a sense, both constants and unknown values are more informative than null values. The proposed repairing strategy aims at exploiting functional dependencies to “infer” more precise information on null values: suppose that $t_1[A]$ is a null value whereas $t_2[A]$ is not, then we replace the null value with $t_2[A]$. We show this idea in the following example.

Example 5.7. Consider the database below.

<i>Employee</i>			<i>Department</i>		
<i>Name</i>	<i>Dept</i>	<i>City</i>	<i>Name</i>	<i>City</i>	<i>Manager</i>
john	cs	rome	cs	rome	carl
bob	cs	milan			

Suppose that the following constraints are defined.

- $fd_1 : Name \rightarrow Dept, City$ defined over $Employee$,
- $fd_2 : Name \rightarrow Manager$ defined over $Department$,
- $fk : Employee[Dept, City] \subseteq Department[Name, City]$.

Thus, a department can be located in different cities and has a unique manager. As the pair $cs, milan$ is missing in the department relation, the database violates fk . By applying the FKC rule we obtain the following database.

<i>Name</i>	<i>Dept</i>	<i>City</i>
<i>john</i>	<i>cs</i>	<i>rome</i>
<i>bob</i>	<i>cs</i>	<i>milan</i>

<i>Name</i>	<i>City</i>	<i>Manager</i>
<i>cs</i>	<i>rome</i>	<i>carl</i>
<i>cs</i>	<i>milan</i>	\perp_1

The obtained database is inconsistent w.r.t. fd_2 since there are two tuples in the department relation regarding the same department cs and containing different information about its manager. As the first tuple states that the manager of cs is $carl$, whereas the second tuple does not say anything, it seems to be reasonable replacing the null value with a more precise information which comes from the first tuple, that is we exploit the functional dependency to “infer” missing information. By applying the FDC rule, the following consistent database is obtained.

<i>Name</i>	<i>Dept</i>	<i>City</i>
<i>john</i>	<i>cs</i>	<i>rome</i>
<i>bob</i>	<i>cs</i>	<i>milan</i>

<i>Name</i>	<i>City</i>	<i>Manager</i>
<i>cs</i>	<i>rome</i>	<i>carl</i>
<i>cs</i>	<i>milan</i>	<i>carl</i>

Observe that unknown values are introduced only in correspondence of attributes appearing in the right-hand side of functional dependencies and, since we consider canonical sets of functional dependencies, unknown values never appear in correspondence of attributes in some left-hand side.

We point out that, since the original database is complete, every null value is introduced either by the FKC rule or by the FDC rule. As it has been discussed above, null values introduced by the FKC rule are no information nulls. Null values introduced by the FDC rule are used just for assigning the same label to different null values, and this does not change their no information interpretation. Thus, we deal only with no information nulls.

It is worth noting that the repairing strategy we propose consists in applying the FDC and FKC rules as long as they are applicable.

Definition 5.4. *Repairing sequence.* Let D be a complete database, FD be a set of functional dependencies and FK be a set of foreign key constraints. A *repairing sequence* of D w.r.t. FD and FK is a (possibly infinite) sequence D_0, \dots, D_j, \dots s.t. $D_0 = D$ and for each $i > 0$ D_i is the database obtained by applying the FDC or the FKC rule to D_{i-1} .

The following proposition states that the proposed repairing process always terminates.

Proposition 5.1. *Let D be a complete database, FD be a set of functional dependencies and FK be a set of foreign key constraints. Each repairing sequence of D w.r.t. FD and FK is finite.* \square

Given a database D , a set FD of functional dependency and a set FK of foreign key constraints, a repairing sequence D, \dots, D_n is *complete* if D_n satisfies $FD \cup FK$ (we call D_n *repaired database*). Clearly, Proposition 5.1 entails that a repaired database always exists.

Corollary 5.1. *Let D be a complete database, FD be a set of functional dependencies and FK be a set of foreign key constraints. There exists a repaired database D_n for D w.r.t. FD and FK .* \square

The following theorem states that, up to renaming of unknown and null values, the repaired database is unique.

Theorem 5.1. *Let D be a complete database, FD be a set of functional dependencies and FK be a set of foreign key constraints. There exists a unique repaired database for D w.r.t. FD and FK (up to renaming of unknown and null values).* \square

As stated in the following theorem, the proposed repairing strategy is polynomial time.

Theorem 5.2. *Let D be a complete database, FD be a set of functional dependencies and FK be a set of foreign key constraints. The repaired database of D w.r.t. FD and FK can be computed in polynomial time.* \square

5.3 Query Answering

In this section we present a semantics of query answering over possibly inconsistent databases. Relying on the results in [56], we show that there exists a class of conjunctive queries which can be evaluated in polynomial time.

In this section we consider only conjunctive queries. A conjunctive query Q is of the form $\exists y \Phi(x, y)$ where Φ is a conjunction of literals (a literal is of the form $p(t_1, \dots, t_n)$ where p is a relation symbol and each t_i is a term, that is a constant or a variable) and x is the set of free variables of Q (x and y are sets of variables).

The proposed semantics of query answering stems from the following observations. The repairing strategy presented in the previous section lead to a consistent database possibly containing unknown and (labeled) null values. Specifically, the so obtained database is an OR-database [56] and thus it represents a set of “possible worlds”, namely the databases which are obtained by replacing every unknown value with a constant of its domain (observe that we treat labeled nulls like standard constants). The “certain answers” to a query over an OR-database are those tuples which can be derived from every

possible world of the database (we observe that, in our case, a possible world can contain labeled nulls; the evaluation of a query over a possible world treats such null values like standard constants). Then, it is natural to define the semantics of query answering over a possibly inconsistent database as the certain answers over its repaired database (Definition 5.5 below). Let D be an OR-database. We denote by $pw(D)$ the set of possible worlds of D .

Definition 5.5. *Consistent Query Answers.* Let D be a complete database, FD be a canonical set of functional dependencies, FK be a set of foreign key constraints and Q be a conjunctive query. Let \bar{D} be the repaired database for D w.r.t. FD and FK . The *consistent answers* to Q on D w.r.t. FD and FK are:

$$Q^c(D, FD \cup FK) = \bigcap_{D \in pw(\bar{D})} Q(D)$$

where $Q(D)$ denotes the result of applying Q over D .

Relying on the results in [56], we show that there exists a class of conjunctive queries whose consistent answers can be computed in polynomial time. The data complexity of queries is considered.

Let us briefly recall the results in [56]. Basically, *OR-Tables* are relations (as presented at the beginning of Section 5.2.1) which do not contain labeled null values. Unknown values are also called *OR-Objects*. Only in correspondence of certain attributes, OR-Tables are allowed to have variables and this is pre-designated by a typing function α . Given a database schema DS , let Att be the set of attribute symbols in DS . The typing function α is defined as $\alpha : Att \rightarrow \{ATOMIC, OR\}$. Those attributes that are mapped to OR are called *OR-Attributes*.

Given a query Q and a literal l in Q , we denote by $VAR(l)$ the set of variables occurring in l .

Definition 5.6. Given a query Q , two literals l_1 and l_2 in Q are *connected* to each other if $VAR(l_1) \cap VAR(l_2) \neq \emptyset$ or, if there exists a literal l_3 in Q such that l_1 is *connected* to l_3 and $VAR(l_3) \cap VAR(l_2) \neq \emptyset$.

Given a database schema DS , a query Q and a literal $l = p(x_1, \dots, x_n)$ in Q , then a variable x_i in l occurring in correspondence of an attribute A is said to *label* A . Moreover, x_i *occurs as OR* in l if it is in correspondence of an OR-Attribute (according to the typing function α for DS).

Definition 5.7. Let DS be a database schema, α be the corresponding typing function, Q be a query and l_1, l_2 be two different literals in Q . Then l_1 *marks* l_2 if there exists a variable $y \in VAR(l_1) \cap VAR(l_2)$, such that y occurs as OR in l_1 or, if there is another literal l_3 in Q such that l_1 *marks* l_3 and l_3 is connected to l_2 .

Marking depends on both the typing function and the way the variables are shared in Q .

Definition 5.8. A query Q is an *acyclic query* if there are no two literals l_1 and l_2 in Q such that l_1 marks l_2 and l_2 marks l_1 . A query which is not acyclic is called *cyclic query*.

Given a database schema DS , a typing function α for it and a query Q , we denote by $MODIFY(\alpha, Q)$ the typing function obtained by modifying α in such a way that every attribute labeled by a free variable in Q is *ATOMIC*. *Proper* conjunctive queries are those queries in which every pair of literals have different relation symbols.

Theorem 5.3. [56] *Let DS be a database schema, α be its typing function, Q be a proper conjunctive query and D be a database instance on DS . Let $\alpha' = MODIFY(\alpha, Q)$. Then, the data complexity of computing the certain answers of Q over D is in *PTIME* iff Q is acyclic with respect to α' . \square*

We recall that null values appearing in repaired databases are treated as constants in query evaluation. Consider a complete database D , a canonical set FD of functional dependencies and a set FK of foreign key constraints. Let \bar{D} be the repaired database for D w.r.t. FD and FK . According to the previous theorem, we can identify proper conjunctive queries which can be evaluated in polynomial time on \bar{D} . As the certain answers computed on \bar{D} coincide with the consistent answers for D w.r.t. FD and FK and \bar{D} can be computed in polynomial time, we can identify queries whose consistent answers over the original database can be computed in polynomial time.

5.4 Discussion

This chapter has proposed a framework for repairing and querying relational databases which may be inconsistent with respect to functional dependencies and foreign key constraints. Specifically, *canonical* sets of functional dependencies have been considered, that is sets of functional dependencies where attributes appearing on the right-hand side cannot appear also on the left-hand side. In order to restore the consistency of inconsistent databases, we have proposed a repairing strategy that performs tuple insertions when foreign key constraints are violated and tuple updates when functional dependency violations occur (tuple deletions are never performed). Since tuple insertions and updates may introduce, respectively, null and unknown values in the database, we have proposed a semantics of constraint satisfaction for databases containing null and unknown values. Our approach always allows us to obtain a unique (up to renaming of unknown and null values) repaired database which can be computed in polynomial time. The result of the repairing technique is an incomplete database (in particular, an OR-database). The consistent query answers over an inconsistent database are the certain answers on the repaired database. The results in [56] on the complexity of query processing in OR-databases allows us to identify conjunctive queries which can be evaluated in polynomial time.

In [60] a semantics of satisfaction of functional and inclusion dependencies in the presence of databases with null values is presented. Here the null value is interpreted as “unknown”. The classical chase procedure is extended to incomplete relations and used to test whether a database satisfies a set of constraints. The axiomatization and the implication problem of functional and inclusion dependencies is studied. No repairing strategy is provided.

The issue of dealing with databases containing null values in the presence of integrity constraints has been considered also in [16]. In this work, null values are also used to repair the original database. The paper considers a wide class of integrity constraints which includes universal integrity constraints, denial constraints, cyclic sets of inclusion dependencies and others. The proposed semantics of constraint satisfaction takes into account *the relevance of the occurrence of a null value* in a relation and is compatible with the way null values are usually treated in commercial database management systems. The notion of repair is that one presented in [4], that is a consistent database instance which minimally differs from the original database.

A Three-Valued Semantics for Querying and Repairing Inconsistent Databases

As it has been show in Section 2.3, computing consistent query answers is in general an intractable problem; polynomial techniques have been proposed only for restricted forms of constraints and queries (see Section 2.2).

In this chapter, a technique for computing “approximate” consistent answers in polynomial time is presented [38, 47]. We consider universal integrity constraints and Datalog queries. The proposed approach is based on a repairing strategy where update operations assigning an *undefined* truth value to the “reliability” of tuples are allowed, along with updates inserting or deleting tuples. The result of a repair can be viewed as a three-valued database satisfying the given constraints. In this regard, a new semantics (namely, *partial semantics*) is introduced for constraint satisfaction in the context of three-valued databases, which aims at capturing the intuitive meaning of constraints under three-valued logic.

It is shown that, in order to compute “approximate” consistent query answers, it suffices to evaluate queries by taking into account a unique repair (called *deterministic* repair), which in some sense “summarizes” all the possible repairs. The so obtained answers are “approximate” in the sense that are safe (true and false atoms in the answers are, respectively, true and false under the classical two-valued semantics), but not complete. We also study some classes of queries and constraints for which the proposed technique is also complete.

6.1 Introduction

In the general case, the problem of computing consistent queries answers is hard. We propose a polynomial-time technique for computing “approximate” consistent query answers in the presence of universal integrity constraints. This class of constraints is relevant in practice, as it suffices to express functional dependencies, join dependencies, and other forms of constraints which are often used to manage data consistency. The proposed approach is based on

a repairing strategy allowing tuple insertions and deletions as updates, as well as update operations assigning an *undefined* truth value to the “reliability” of tuples. The result of a repair can be viewed as a three-valued database which satisfies the specified constraints. In this regard, we introduce a new semantics for constraint satisfaction in the context of three-valued databases, which aims at capturing the intuitive meaning of constraints under three-valued logic.

We show that, in order to compute “approximate” consistent query answers, it suffices to evaluate queries by taking into account a unique repair (called *deterministic* repair), which in some sense “summarizes” all the possible repairs. The so obtained answers are “approximate” in the sense that they are safe (true and false atoms in the answers are, respectively, true and false under the classical two-valued semantics), but not complete. Thus, we show that the deterministic repair can be computed by evaluating the (unique) stable model of a logic program, whose rules are a suitable rewriting of the integrity constraints which takes into account the content of the source database.

The novelty of the proposed approach consists in (i) the definition of a three-valued semantics for constraint satisfaction under three-valued databases, (ii) the definition of (three-valued) repairs under such a semantics, (iii) the definition of consistent query answers for Datalog queries under the proposed semantics. More specifically:

- We consider databases whose atoms may be either *true* or *undefined* (missing tuples are assumed to be *false*). The reason for considering three-valued databases is that in the computation of repairs under three-valued semantics, some of the atoms may not be assumed to be *true* or *false* and they are assumed to be *undefined*.
- As databases may be three-valued, we propose a different semantics for the satisfaction of integrity constraints which for standard (two-valued) databases coincides with the classical semantics.
- We propose three-valued repairs consisting of update operations which make the truth value of database atoms *true*, *false* or *undefined*. We show that the set of three-valued repairs defines a lower semi-lattice whose top elements are standard (two-valued) repairs and whose bottom element defines the deterministic repair.
- We show that “approximate” consistent answers can be computed by taking into account only the database obtained by applying the deterministic repair to the source (inconsistent) database. This evaluation is sound (true and false atoms in the answers are, respectively, true and false under the classical two-valued semantics), but not complete.
- Finally, we show that deterministic repairs and query answers can be computed in polynomial time, by showing that a logic program whose perfect model corresponds to the deterministic repair can be obtained by suitably rewriting the integrity constraints associated with the database.

The rest of the chapter is organized as follows. In Section 6.2, notation and terminology used in this chapter are presented. In Section 6.3 a constraint satisfaction semantics for three-valued databases, called *partial*, is proposed. This section also introduces the definition of repair and of consistent query answers under the partial semantics. Section 6.5 presents a technique which allows us to derive logic rules from constraints and to compute deterministic repairs by considering the fixpoint of such a program. Finally, in Section 6.6 related works are discussed and conclusions are drawn.

6.2 Notation and Terminology

Three-valued semantics. Notions of interpretation, minimal model and stable model can be extended to three-valued (or partial) semantics, where every atom may be either *true* or *false* or *undefined*.

For any three-valued interpretation I , I^+ (resp. I^- , I^u) denotes the set of *true* (resp. *false*, *undefined*) (ground) atoms in I . The value of a ground atom A w.r.t. an interpretation I is denoted as $value_I(A)$ (this is extended to literals, conjunctions and disjunctions). In this context, truth value order $true > undefined > false$ is assumed, and $\neg undefined = undefined$.

Any partial model M can be defined by means of two sets of atoms (for instance, the *true* and *false* atoms M^+ and M^-). A model $M = \langle M^+, M^- \rangle$ for a program P is minimal if there is no model $N = \langle N^+, N^- \rangle$ such that $N^+ \subseteq M^+$, $N^- \supseteq M^-$ and $N \neq M$.

Also in the case of three-valued semantics, positive programs admit a unique minimal model. Given a program P , P^M denotes the ground positive program obtained from $ground(P)$ by replacing every negative body literal $\neg A$ with its truth value w.r.t. M . An interpretation I is a *partial stable model* of P if and only if $I \in \mathcal{MM}(P^M)$. The set of partial stable models of a program P is denoted by $\mathcal{PSM}(P)$, and the intersection of all partial stable models (equal to $\langle \bigcap_{M \in \mathcal{PSM}(P)} M^+, \bigcap_{M \in \mathcal{PSM}(P)} M^- \rangle$) is a partial stable model also known as the *well-founded* model (denoted by $\mathcal{WFM}(P)$) [74].

Queries. Given a set D of facts and a predicate symbol p , then $D[p]$ denotes the set of facts in D whose predicate symbol is p . Queries are expressed by means of Datalog programs. More formally, a (*Datalog*) *query* q is a pair (g, P) where g is a predicate symbol, called the *query goal*, and P is a Datalog program. The answer to a Datalog query $q = (g, P)$ over a database D is $M[g]$, where M is the unique minimal model in $\mathcal{MM}(P \cup D)$, and will be denoted as $q(D)$.

Integrity constraints. Given a universal integrity constraint ic of the form

$$\forall X[L_1 \wedge \dots \wedge L_m \wedge \phi \rightarrow L_{m+1} \vee \dots \vee L_n]$$

where the L_i 's are literals and ϕ is a conjunction of built-in atoms, the conjunction on the left-hand side of the implication is the *body* of ic , denoted by $Body(ic)$, whereas the disjunction on the right-hand side of the implication is called the *head*, denoted by $Head(ic)$.

A *ground instance* of ic w.r.t. a database D is obtained from ic by replacing every variable with a constant in the Herbrand universe U_D and eliminating the universal quantifier. The set of ground instances of ic w.r.t. D is denoted as $ground_D(ic)$. Given a set IC of universal constraints and a database D , we define the ground instance of IC w.r.t. D as $ground_D(IC) = \bigcup_{ic \in IC} ground_D(ic)$. Clearly, for any IC and D , the cardinality of $ground_{\mathcal{DB}}(IC)$ is polynomial in the size of D . For the sake of simplicity, whenever D is understood from the context, we simply write $ground(IC)$ instead of $ground_D(IC)$.

We assume that constraints are written as an implication with empty head. In the following, we consider ground constraints without built-in atoms, as, when evaluating the ground instance of a constraint, built-in atoms evaluating to true can be disregarded, and constraints with built-in atoms evaluating to false can be deleted (as they are trivially satisfied).

Finally, given a set of constraints IC , we assume that $ground(IC)$ does not contain any pair of constraints ic_1, ic_2 such that the set of literals occurring in ic_1 is a proper subset of the literals occurring in ic_2 : in fact, in this case, the satisfaction of ic_1 implies the satisfaction of ic_2 , thus the latter can be disregarded.

Repairing databases. In this chapter, we use the term *repair* for a minimal (under set inclusion) set of tuple insertions/deletions which lead a database to a consistent state. *Repaired databases* are consistent databases obtained from the source database by applying a repair. Update operations can be represented by *update atoms* of the form $a(X)^+$ (*inserting atom*) or $a(X)^-$ (*deleting atom*). Intuitively, a ground atom $a(t)^+$ states that $a(t)$ must be inserted into the database, whereas a ground atom $a(t)^-$ states that $a(t)$ must be deleted from the database. Given a set Up of ground update atoms, we define the sets: $Up^+ = \{a(t) \mid a(t)^+ \in Up\}$, $Up^- = \{a(t) \mid a(t)^- \in Up\}$. We say that Up is *consistent* if it does not contain two “conflicting” update atoms $a(t)^+$ and $a(t)^-$ (i.e., if $Up^+ \cap Up^- = \emptyset$). Given a database D and a consistent set of update atoms Up , we denote as $Up \circ D$ the updated database $D \cup Up^+ - Up^-$.

Definition 6.1. Given a database D and a set IC of universal constraints, a *repair* for $\langle D, IC \rangle$ is a consistent set of update atoms R such that: 1) $R \circ D \models IC$ and 2) there is no consistent set of update atoms $Up \subset R$ such that $Up \circ D \models IC$.

The set of the repairs for a database D w.r.t a set IC of universal constraints is denoted as $repairs(D, IC)$. It is worth noting that $repairs(D, IC)$ is not

empty for any set of universal constraints: in fact, deleting all the tuples from a database leads to a consistent state.

Definition 6.2. Given a query $q = (g, P)$, the consistent answer to q (CQA) on database D w.r.t. a set IC of universal constraints, denoted as $q(D, IC)$, is the triplet of sets $q(D, IC)^+$, $q(D, IC)^-$, $q(D, IC)^u$, where:

- $q(D, IC)^+ = \{g(t) \mid g(t) \in \bigcap_{R \in \text{repair}(D, IC)} q(R \circ D)\};$
- $q(D, IC)^- = \{g(t) \mid g(t) \in B_{D \cup P} \wedge g(t) \notin \bigcup_{R \in \text{repairs}(D, IC)} q(R \circ D)\};$
- $q(D, IC)^u = \{g(t) \mid g(t) \in B_{D \cup P} - (q(D, IC)^+ \cup q(D, IC)^-)\}.$

In the previous definition $q(D, IC)^+$ is the set of g -tuples which are true, that is each tuple $g(t)$ in $q(D, IC)^+$ is in $q(RD)$ for each repaired database RD ; $q(D, IC)^-$ is the set of g -tuples which are false, that is for each tuple $g(t)$ in $q(D, IC)^-$ there is no repaired database RD such that $g(t)$ is in $q(RD)$; finally, $q(D, IC)^u$ is the set of undefined tuples, that is tuples which are neither true nor false.

In the following section, we will introduce a new framework for repairing and querying inconsistent data. Our approach is based on a different notion of repair (namely, *partial repair*), enabling tuples to be inserted, deleted, or made undefined. In order to distinguish the proposed notion of repair from that of Definition 6.1 (based on insert/delete operations only), we will refer to the latter as *total repair*. Likewise, we will refer to the consistent answer evaluated by considering all the databases resulting from total repairs as *classical consistent query answer*.

6.3 Partial Repairs

The proposed framework is based on the notion of *partial repair*, which extends the total repairs presented in the previous section. Specifically, a partial repair makes a database consistent by making tuples true, false, or undefined.

As applying a partial repair to a database results in a three-valued database, we introduce a new (three-valued) semantics of constraint satisfaction, which for total repairs (and two-valued databases) coincides with the classical semantics, and aims at preserving different aspects of the two-value semantics when moving to the three-valued logic.

In the following, given a three-valued database D , we denote with D^+ (resp. D^u , D^-) the set of tuples which are *true* (resp. *undefined*, *false*) in D .

6.3.1 Constraint Satisfaction Under Partial Semantics

The semantics of constraint satisfaction can be trivially extended from two-valued databases to three-valued databases: given a three-valued database D and a ground constraint ic , D satisfies ic , denoted as $D \models ic$, if

$value_{\mathcal{DB}}(Head(ic)) \geq value_{\mathcal{DB}}(Body(ic))$. This semantics of constraint satisfaction will be referred to as *three-valued semantics*.

However, there are some aspects of the two-valued semantics which are not preserved by the above-mentioned extension to the three-valued logic. Specifically, under the two-valued semantics, literals appearing in a constraint can be moved from the head to the body and vice versa without modifying the truth value of the constraint, but this property does not hold under the three-valued semantics.

Example 6.1. Consider the database D consisting of $D^+ = \{\mathbf{a}\}$ and $D^u = \{\mathbf{b}, \mathbf{c}\}$. This database satisfies the constraint $ic = \mathbf{a} \wedge \mathbf{b} \supset \mathbf{c}$, but does not satisfy the constraint $ic' = \mathbf{b} \wedge \neg \mathbf{c} \supset \neg \mathbf{a}$ which is derived from ic by moving \mathbf{a} to the head and \mathbf{c} to the body. In fact, $D \models ic$ as $value_{\mathcal{DB}}(Body(ic)) = value_{\mathcal{DB}}(Head(ic)) = \text{undefined}$, while $D \not\models ic'$ as $value_{\mathcal{DB}}(Body(ic')) = \text{undefined}$ and $value_{\mathcal{DB}}(Head(ic')) = \text{false}$.

In some sense, the above-reported example shows that, given an integrity constraint ic defined on a two-valued database, the interpretation of ic in the three-valued context may not capture some aspects of the meaning of the original constraint. Thus, in the following, we introduce a new semantics (namely, *partial semantics*) for constraint satisfaction in the presence of three-valued databases. As it will be clearer in the following, the introduction of this semantics is at the basis of our framework for extracting reliable information from inconsistent databases: we will show that, given an inconsistent two-valued database, there is a “partial” repair which yields a consistent three-valued database (where consistency is defined by partial semantics) and which summarizes all the possible total repairs, enabling “approximate” query answers to be computed.

Under the two valued semantics, a (denial) constraint whose body is *false* is satisfied. Therefore, in order to satisfy a constraint ic , a literal appearing in ic must be *false* if all other literals are *true*. For instance, a constraint of the form $L \supset$ states that the literal L must be *false*.

Given a set IC of ground constraints, we define the set $\Psi_{IC} = \mathcal{W}_{IC}^{\infty}(\emptyset)$ where

$$\mathcal{W}_{IC}(T) = \{\neg L \mid \exists ic = L \wedge L_1 \wedge \dots \wedge L_n \supset \in IC \text{ s.t. } L_1, \dots, L_n \in T\}$$

Intuitively, Ψ_{IC} defines the set of literals which must be *true* in order to satisfy IC (that is, a literal L in Ψ_{IC} is *true* in any database which satisfies IC). Once Ψ_{IC} has been evaluated, the set IC of ground constraints can be “simplified” performing the following steps:

1. for each constraint $ic \in IC$, every literal L occurring in ic such that either L or $\neg L$ is in Ψ_{IC} can be replaced with the corresponding truth value;
2. constraints in IC containing the truth value *false* in the body can be deleted, whereas the truth value *true* can be deleted from the body conjunctions.

The reduced set of constraints obtained from IC by performing the sequence of steps explained above will be denoted as $Red(IC)$.

Example 6.2. Consider the set $IC = \{a \supset, \neg a \wedge b \supset, \neg b \wedge c \wedge d \supset\}$ of (ground) integrity constraints. The set Ψ_{IC} is equal to $\{\neg a, \neg b\}$. The reduced set of constraints contains only the constraint $c \wedge d \supset$ (the first two constraints are deleted as a and b are replaced with *false*).

Observe that, given a set IC of ground constraints, every constraint in $Red(IC)$ contains at least two literals and no built-in literal.

Definition 6.3. Let ic be a ground constraint of the form $L_1 \wedge \dots \wedge L_n \supset$ with $n > 1$. We define the set of *extended constraints* derived from ic as:

$$ext(ic) = \left\{ \bigwedge_{L_j \in T} L_j \supset \bigvee_{L_j \in \{L_1, \dots, L_n\} - T} \neg L_j \mid T \subset \{L_1, \dots, L_n\} \wedge |T| > 0 \right\}$$

Moreover, given a set IC of ground integrity constraints of the form above, we define the set $ext(IC) = \cup_{ic \in IC} ext(ic)$.

Basically, given a constraint ic , $ext(ic)$ is the set of constraints which are derived from ic by moving body literals to the head so that both the head and the body are not empty.

The following definition states when a set of constraints is satisfiable under partial semantics.

Definition 6.4. Let D be a database and IC be a set of constraints. Then, D satisfies IC under partial semantics (denoted as $D \models_{\mathcal{P}_S} IC$) if

- $D \models \Psi_{ground(IC)}$ (i.e., $\Psi_{ground(IC)} \subseteq D^+ \cup \{\neg A \mid A \in D^-\}$), and
- $D \models ext(ground(IC))$

Since under the classical three-valued semantics a (ground) constraint ic in $Red(ground(IC))$ can change its truth value by moving literals from the body to the head and vice versa (see Example 6.1), in the definition above we say that a database D satisfies ic under partial semantics if D satisfies (under classical three-valued semantics) all constraints which can be derived from ic by moving any set of literals to the head (provided that both the head and the body of the derived constraints are not empty).

Example 6.3. Consider the following set IC of ground integrity constraints:

$$\begin{aligned} & \mathbf{a} \supset \\ & \neg \mathbf{a} \wedge \mathbf{b} \supset \\ & \neg \mathbf{a} \wedge \mathbf{c} \wedge \neg \mathbf{d} \supset \\ & \mathbf{b} \wedge \mathbf{e} \wedge \neg \mathbf{f} \supset \\ & \neg \mathbf{g} \wedge \mathbf{h} \supset \end{aligned}$$

Thus $\Psi_{IC} = \{\neg \mathbf{a}, \neg \mathbf{b}\}$, $Red(IC)$ is as follows

$$\begin{aligned} c \wedge \neg d \supset \\ \neg g \wedge h \supset \end{aligned}$$

and $\text{ext}(\text{Red}(IC))$ is as follows

$$\begin{array}{ll} c \supset d & \neg g \supset \neg h \\ \neg d \supset \neg c & h \supset g \end{array}$$

The database D consisting of $D^+ = \emptyset$, $D^- = \{\mathbf{a}, \mathbf{b}, \mathbf{e}, \mathbf{f}\}$, $D^u = \{\mathbf{c}, \mathbf{d}, \mathbf{g}, \mathbf{h}\}$ satisfies IC under partial semantics as $D \models \Psi_{IC}$ and $D \models \text{ext}(\text{Red}(IC))$. The same holds for the database D_1 consisting of $D_1^+ = \emptyset$, $D_1^- = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}\}$, $D_1^u = \{\mathbf{g}, \mathbf{h}\}$, as well as the database D_2 consisting of $D_2^+ = \emptyset$, $D_2^- = \{\mathbf{a}, \mathbf{b}, \mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}$, $D_2^u = \{\mathbf{c}, \mathbf{d}\}$.

Proposition 6.1. *Let D be a database and IC a set of ground full constraints. Then, $D \models \text{ext}(\text{Red}(IC))$ iff, for each constraint $ic = L_1 \wedge \dots \wedge L_n \supset$ in $\text{Red}(IC)$, either (i) some literal L_i is false w.r.t. D , or (ii) all literals L_1, \dots, L_n are undefined w.r.t. D .*

Proof. First of all, we recall that, for every constraint in $\text{Red}(IC)$ (and, consequently, in $\text{ext}(\text{Red}(IC))$), the number of literals is strictly greater than 1.

We first show that if, for each $ic \in \text{Red}(IC)$, either (i) or (ii) holds, then $D \models \text{ext}(\text{Red}(IC))$. Clearly, for every $ic \in \text{Red}(IC)$ such that there is some literal L_i which is false w.r.t. D , it holds that every constraint in $\text{ext}(\text{Red}(ic))$ is satisfied, since either its body is false (as it contains L_i) or its head is true (as it contains $\neg L_i$). Analogously, for each $ic \in \text{Red}(IC)$ whose literals are undefined, every constraint in $\text{ext}(\text{Red}(ic))$ is satisfied, as both its body and head evaluate to undefined.

We now show the inverse implication, reasoning by contradiction. Assume that $D \models \text{ext}(\text{Red}(IC))$ and there is an $ic \in \text{Red}(IC)$ where some literal L_i is true w.r.t. D and all the other literals $L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n$ are either true or undefined. This implies that the constraint $L_1, \dots, L_{i-1}, L_{i+1}, \dots, L_n \supset \neg L_i$ of $\text{ext}(\text{Red}(ic))$ is not satisfied in D , thus contradicting $D \models \text{ext}(\text{Red}(IC))$. \square

Example 6.4. Consider the set $IC = \{\mathbf{a} \wedge \mathbf{b} \supset, \mathbf{b} \wedge \neg \mathbf{c} \supset\}$ of ground integrity constraints. Then, $\text{ext}(IC) = \{\mathbf{a} \supset \neg \mathbf{b}, \mathbf{b} \supset \neg \mathbf{a}, \mathbf{b} \supset \mathbf{c}, \neg \mathbf{c} \supset \neg \mathbf{b}\}$. These constraints are satisfied iff either (i) \mathbf{b} is *false*, or (ii) \mathbf{a} is *false* and \mathbf{c} is *true*, or (iii) \mathbf{a} is *false* and both \mathbf{b} and \mathbf{c} are *undefined*, or (iv) \mathbf{c} is *true* and both \mathbf{b} and \mathbf{a} are *undefined*, or (v) all atoms \mathbf{a} , \mathbf{b} and \mathbf{c} are *undefined*.

Observe that the cardinality of $\text{ext}(ic)$ is exponential in the number of literals appearing in ic . We now show that it is sufficient to consider a number of derived constraints equal to the number of database literals appearing in ic .

Definition 6.5. Let ic be a ground constraint of the form $L_1 \wedge \dots \wedge L_n \supset$ with $n > 0$. We define the *single-head extended constraints* derived from ic as

$$sh-ext(ic) = \{L_1 \wedge \dots \wedge L_{i1} \wedge L_{i+1} \wedge \dots \wedge L_n \supset \neg L_i \mid i \in [1..n]\}$$

Moreover, given a set IC of ground (denial) constraints, we define $sh-ext(IC) = \cup_{ic \in IC} sh-ext(ic)$.

Basically, given a constraint ic , $sh-ext(ic)$ is the set of constraints which are derived from ic by moving exactly one body literal to the head.

Proposition 6.2. *Given a database D and a set IC of ground full constraints, then $D \models ext(Red(IC))$ iff $D \models sh-ext(Red(IC))$.*

Proof. The implication $D \models ext(Red(IC)) \Rightarrow D \models sh-ext(Red(IC))$ trivially follows from the fact that $sh-ext(Red(IC)) \subseteq ext(Red(IC))$. We now show that the inverse implication holds too by proving its contrapositive (that is, $D \not\models ext(Red(IC)) \Rightarrow D \not\models sh-ext(Red(IC))$).

Let ic be a constraint in $ext(Red(IC))$ such that $D \not\models ic$. If ic is of the form $L_1 \wedge \dots \wedge L_{n-1} \supset L_n$, then it is the case that $ic \in sh-ext(Red(IC))$, thus $D \not\models sh-ext(Red(IC))$ holds too. Otherwise, the head of ic consists of at least two literals, that is ic is of the form $L_1 \wedge \dots \wedge L_m \supset L_{m+1} \vee \dots \vee L_n$, where $n - m > 1$. As $D \not\models ext(Red(IC))$, two cases may occur: 1) the body of ic evaluates to true, whereas the head of ic evaluates to either false or undefined; 2) the body of ic evaluates to undefined, whereas the head of ic evaluates to false.

If case 1) holds, it means that all the body literals of ic evaluate to true, whereas the head literals to false or undefined. Consider the constraint ic' in $sh-ext(Red(IC))$ obtained from ic by moving the head literals to the body and one of the body literals to the head. It is easy to see that $D \not\models ic'$ (as the body of ic' evaluates to either true or undefined, while its head evaluates to false), thus $D \not\models sh-ext(Red(IC))$.

Analogously, if case 2) holds, then at least one body literal of ic evaluate to undefined, the other body literals to either true or undefined, whereas all the head literals evaluate to false. Consider the constraint ic'' in $sh-ext(Red(IC))$ obtained from ic by moving all the head literals to the body and the undefined body literals to the head. It is easy to see that $D \not\models ic''$ (as the body of ic'' evaluates to true, while its head evaluates to undefined), thus $D \not\models sh-ext(Red(IC))$. \square

Therefore, in the definition of partial semantics (Definition 6.4), instead of considering the set $ext(Red(ground(IC)))$, whose size is exponential in the number of database literals appearing in the body of constraints, we can consider $sh-ext(Red(ground(IC)))$, whose size is linear in the number of ground constraints and the number of literals appearing in the body of constraints. Consequently, we have that $D \models_{\mathcal{PS}} IC$ if (i) $D \models \Psi_{ground(IC)}$, and (ii) $D \models sh-ext(Red(ground(IC)))$.

6.3.2 Repairing Databases

An update atom is of the form $a(X)^+$ or $a(X)^-$ or $a(X)^u$. As said before, a ground update atom $a(t)^+$ states that $a(t)$ will be inserted into the database, whereas $a(t)^-$ states that $a(t)$ will be deleted from the database. The meaning of a ground update atom $a(t)^u$ is that $a(t)$ is made undefined.

Definition 6.6. A database update Up for a database D is a set of update atoms such that for each $a(t)^v \in Up$

- there is no atom $a(t)^{v'} \in Up$ such that $v' \neq v$, and
- the atom $a(t)$ does not belong to D^v . □

Basically, the first condition stated in Definition 6.6 means that Up cannot perform two distinct update operations on the same database atom. The second condition means that every update operation in Up must change the truth value of the database atom on which it is applied.

In the following we also use the notations $Up^+ = \{a(t) | a(t)^+ \in Up\}$, $Up^- = \{a(t) | a(t)^- \in Up\}$ and $Up^u = \{a(t) | a(t)^u \in Up\}$ to denote the set of atoms which are made, respectively, *true*, *false*, and *undefined* by Up .

Given a database D and a database update Up for D , the application of Up to D , denoted by $Up \circ D$, gives the following sets: (i) $(Up \circ D)^+ = (D^+ - Up^u - Up^-) \cup Up^+$ and (ii) $(Up \circ D)^u = (D^u - Up^+ - Up^-) \cup Up^u$. Observe that the set $(Up \circ D)^- = B_{\mathcal{DB}} - (Up \circ D)^+ - (Up \circ D)^u$ is also equal to $(D^- - Up^+ - Up^u) \cup Up^-$.

Definition 6.7. Given a database D and a set IC of constraints, a *partial repair* for $\langle D, IC \rangle$ is a database update R for D s.t. (i) $(R \circ D) \models_{\mathcal{PS}} IC$, and (ii) there is no database update Up s.t. $(Up \circ D) \models_{\mathcal{PS}} IC$ and $Up \subset R$.

The definition above states that a repair for a database and a set of integrity constraints, under partial semantics, is a minimal database update which makes the database consistent. Repaired databases are consistent databases derived from the source database by applying repairs over it. Given a database D and a set IC of integrity constraints, the set of all possible partial repairs for $\langle D, IC \rangle$ is denoted as $repairs_{\mathcal{PS}}(D, IC)$. Moreover, $repaired_{\mathcal{PS}}(D, IC)$ denotes the set of all possible repaired databases for $\langle D, IC \rangle$, i.e. $repaired_{\mathcal{PS}}(D, IC) = \{R \circ D \mid R \in repairs_{\mathcal{PS}}(D, IC)\}$.

Given two repairs R_1 and R_2 , we say that $R_1 \sqsubseteq R_2$ if $R_1^+ \subseteq R_2^+$ and $R_1^- \subseteq R_2^-$. The same notation will be used for databases, namely, given two databases D_1 and D_2 , we say that $D_1 \sqsubseteq D_2$ if $D_1^+ \subseteq D_2^+$ and $D_1^- \subseteq D_2^-$.

Lemma 6.1. Let D be a database and IC a set of integrity constraints over D . For each two repairs R_i and R_j in $repairs_{\mathcal{PS}}(D, IC)$, there exists a repair R_k in $repairs_{\mathcal{PS}}(D, IC)$ s.t. $R_k \sqsubseteq R_i$ and $R_k \sqsubseteq R_j$.

Proof. For each two repairs R_i, R_j in $repairs_{\mathcal{PS}}(D, IC)$, we build a database update R_k as follows:

1. $R_k := R_i \cap R_j$. Observe that, if R_i and R_j are two distinct repairs, there exists some constraint ic in $Red(ground(IC))$ such that $R_k \circ D \not\models sh-ext(ic)$.
2. for each constraint $ic : A_1 \wedge \dots \wedge A_m \wedge \neg A_{m+1} \wedge \dots \wedge \neg A_n \supset$ in $Red(ground(IC))$ such that $R_k \circ D \not\models sh-ext(ic)$,
 $R_k := R_k - \{A_1^+, \dots, A_n^+, A_1^-, \dots, A_n^-\} \cup \{A_1^u, \dots, A_n^u\}$
3. repeat Step 2 until $R_k \circ D \models sh-ext(Red(ground(IC)))$.

It is easy to see that the database update R_k makes D consistent. If R_k is minimal, then it is a repair and, moreover, it holds that $R_k \sqsubseteq R_i$ and $R_k \sqsubseteq R_j$ by construction. If R_k is not minimal, there exists a repair R'_k s.t. $R'_k \subset R_k$; in this case it holds that $R'_k \sqsubseteq R_i$ and $R'_k \sqsubseteq R_j$. \square

Theorem 6.1. *Given a database D and a set IC of integrity constraints over D , then $\langle repairs_{\mathcal{PS}}(D, IC), \sqsubseteq \rangle$ is a lower semi-lattice.*

Proof. Obviously, $\langle repairs_{\mathcal{PS}}(D, IC), \sqsubseteq \rangle$ is a partial order. From Lemma 6.1 we have that for each two repairs R_i and R_j there must be a repair R_k such that $R_k \sqsubseteq R_i$ and $R_k \sqsubseteq R_j$. Therefore, the set of partial repairs defines a lower semi-lattice. \square

It is easy to see that the set of top elements of the semi-lattice $\langle repairs_{\mathcal{PS}}(D, IC), \sqsubseteq \rangle$ coincides with the set of total repairs $repairs(D, IC)$; the repair defining the bottom element will be said to be *deterministic* and will be denoted as $R_{det}(D, IC)$. The database obtained by applying the deterministic repair to D will be called *deterministic repaired database* and will be denoted as $D_{det}(D, IC)$. In the following, whenever D and IC are understood, we will write R_{det} instead of $R_{det}(D, IC)$, as well as D_{det} instead of $D_{det}(D, IC)$.

Example 6.5. Consider the integrity constraints of Example 6.4 and the (two-valued) database $D = \{\mathbf{a}, \mathbf{b}\}$. There are five partial repairs for $\langle D, IC \rangle$: $R_1 = \{\mathbf{b}^-\}$, $R_2 = \{\mathbf{a}^-, \mathbf{c}^+\}$, $R_3 = \{\mathbf{a}^-, \mathbf{b}^u, \mathbf{c}^u\}$, $R_4 = \{\mathbf{c}^+, \mathbf{b}^u, \mathbf{a}^u\}$ and $R_5 = \{\mathbf{a}^u, \mathbf{b}^u, \mathbf{c}^u\}$. R_1 and R_2 are total repairs, whereas R_5 is the deterministic repair.

Corollary 6.1. *Given a database D and a set IC of integrity constraints over D , then $\langle repaired_{\mathcal{PS}}(D, IC), \sqsubseteq \rangle$ is a lower semi-lattice whose top elements are two-valued databases and whose bottom element is the deterministic repaired database.*

Proof. Straightforward from Theorem 6.1. \square

The following theorem gives an insight on how the deterministic repaired database summarizes all the two-valued repaired databases.

Theorem 6.2. *Given a database D and a set IC of integrity constraints over D , an atom A is true (resp. false) with respect to $\langle D, IC \rangle$ if A is true (resp.*

false) w.r.t. the deterministic repaired database.

Proof. The two-valued repaired databases define the top elements of the semi-lattice. Therefore, for every two-valued repaired database D , it holds that $D_{det}^+ \subseteq D^+$ and $D_{det}^- \subseteq D^-$ (Corollary 6.1). Hence, all true and false atoms in D_{det} are also true and false, respectively, in all the two-valued repaired databases. \square

It is worth noting that the implication of Theorem 6.2 does not hold in the opposite direction. In fact, if an atom A is either true or false in all the two-valued repaired databases, it can be the case that A is undefined in the deterministic repaired database, as shown in the following example.

Example 6.6. Consider the database D consisting of $D^+ = \{\mathbf{a}, \mathbf{b}\}$ and the set IC consisting of the following integrity constraints:

$$\begin{aligned} \mathbf{a}, \mathbf{b} &\supset \\ \mathbf{a}, \neg\mathbf{b} &\supset \\ \neg\mathbf{a}, \mathbf{b} &\supset \end{aligned}$$

It is easy to see that the set of two-valued repaired databases for $\langle D, IC \rangle$ consists of $D_1 = \emptyset$ only, while the deterministic repaired database D_{det} consists of: $D_{det}^+ = \emptyset$; $D_{det}^- = \emptyset$; $D_{det}^u = \{\mathbf{a}, \mathbf{b}\}$. Therefore, \mathbf{a} and \mathbf{b} are false in all the two-valued repaired databases, but undefined in the deterministic repaired database.

6.4 Query Answering

The computation of a query $q = (g, P)$ over a three-valued database D can be obtained by considering the well-founded semantics of the program $P \cup P^u \cup D^+$, where P^u is the set of rules $\{a(w) \leftarrow \neg a(w) \mid a(w) \in D^u\}$, which are used to derive the atoms in D^u as *undefined*.

Thus, given a Datalog query $q = (g, P)$ and a database D , the answer of q over D , denoted by $q(D)$, gives three sets denoted as $q(D)^+$, $q(D)^u$ and $q(D)^-$. These sets contain, respectively, the g -tuples which are derived as *true*, *undefined* and *false* by applying $P \cup P^u$ over D^+ , i.e., $q(D)^+ = \mathcal{WFM}(P \cup P^u \cup D^+)^+[g]$, $q(D)^u = \mathcal{WFM}(P \cup P^u \cup D^+)^u[g]$, and $q(D)^- = B_{P \cup D}[g] - q(D)^+ - q(D)^u$.

We introduce an extension of the notion of classical consistent query answer (Definition 6.2) to our framework, where the consistent information is determined by taking into account partial repairs (rather than total repairs only).

Definition 6.8. Given a database D , a set IC of integrity constraints and a query $q = (g, P)$, the *consistent answer to q on $\langle D, IC \rangle$ under partial semantics*, denoted as $q_{\mathcal{PS}}(D, IC)$, is the triplet $q_{\mathcal{PS}} = \langle q_{\mathcal{PS}}(D, IC)^+, q_{\mathcal{PS}}(D, IC)^-, q_{\mathcal{PS}}(D, IC)^u \rangle$, where:

$$\begin{aligned}
q_{\mathcal{PS}}(D, IC)^+ &= \bigcap_{R \in \text{repairs}_{\mathcal{PS}}(D, IC)} q(R \circ D)^+ \\
q_{\mathcal{PS}}(D, IC)^- &= \bigcap_{R \in \text{repairs}_{\mathcal{PS}}(D, IC)} q(R \circ D)^- \\
q_{\mathcal{PS}}(D, IC)^u &= B_{P \cup D} - q_{\mathcal{PS}}(D, IC)^+ - q_{\mathcal{PS}}(D, IC)^-
\end{aligned}$$

□

The following theorem states that computing the consistent answer of a query q under partial semantics is equivalent to evaluating q on the deterministic repaired database.

Theorem 6.3. *Let D be a database, IC a set of integrity constraints, $q = (g, P)$ a Datalog query, and D_{det} the deterministic repaired database for $\langle D, IC \rangle$. Then, $q_{\mathcal{PS}}(D, IC) = q(D_{det})$.*

Proof. As the repaired databases form a lower semi-lattice under relation \sqsubseteq (Theorem 6.1) whose bottom element is the deterministic repaired database, for every repaired database D it holds that $D_{det}^+ \subseteq D^+$ and $D_{det}^- \subseteq D^-$. Since Datalog queries are monotonic under three-valued logic, $D_{det}^+ \subseteq D^+$ implies $q(D_{det})^+ \subseteq q(D)^+$, and $D_{det}^- \subseteq D^-$ implies $q(D_{det})^- \subseteq q(D)^-$, for any repaired database D . Therefore, $\bigcap_{R \in \text{repairs}_{\mathcal{PS}}(D, IC)} q(R \circ D)^+ = q(R_{det} \circ D)^+$ and $\bigcap_{R \in \text{repairs}_{\mathcal{PS}}(D, IC)} q(R \circ D)^- = q(R_{det} \circ D)^-$. Finally, it is easy to see that $q_{\mathcal{PS}}(D, IC)^u = q(R_{det} \circ D)^u$. □

In the rest of this work, the answer of a query evaluated on the deterministic repaired database only will be referred to as *deterministic answer*, and will be denoted as $q_{det}(D, IC)$. The following theorem states that the deterministic answer to a query q provides a “sound” evaluation of the classical consistent answer to q (which takes into account the two-valued repaired databases only): atoms which are true (resp., false) in $q_{det}(D, IC)$ are true (resp., false) in the classical consistent answer to q on $\langle D, IC \rangle$ too.

Theorem 6.4. *[Soundness] Let D be a two-valued database, IC a set of integrity constraints over D , and $q = (g, P)$ a Datalog query. Then, $q_{det}(D, IC) \sqsubseteq q(D, IC)$.*

Proof. As $q_{det}(D, IC)^+ = \bigcap_{R \in \text{repairs}_{\mathcal{PS}}(D, IC)} q(R \circ D)^+$, $q(D, IC)^+ = \bigcap_{R \in \text{repairs}(D, IC)} q(R \circ D)^+$, and $\text{repairs}(D, IC) \subseteq \text{repairs}_{\mathcal{PS}}(D, IC)$, it holds that $q_{det}(D, IC)^+ \subseteq q(D, IC)^+$. Analogously, as $q_{det}(D, IC)^- = \bigcap_{R \in \text{repairs}_{\mathcal{PS}}(D, IC)} q(R \circ D)^-$, $q(D, IC)^- = \bigcap_{R \in \text{repairs}(D, IC)} q(R \circ D)^-$, and $\text{repairs}(D, IC) \subseteq \text{repairs}_{\mathcal{PS}}(D, IC)$, it also holds that $q_{det}(D, IC)^- \subseteq q(D, IC)^-$. Therefore, $q_{det}(D, IC) \sqsubseteq q(D, IC)$. □

Observe that Theorem 6.4 can be viewed as a generalization of Theorem 6.2 to the case that the query goal is not a base predicate. Hence, it is easy to see that, in general, the implication of Theorem 6.4 does not hold in the opposite direction (as shown for Theorem 6.2 – see Example 6.6). That is, evaluating a query on D_{det} is not a complete strategy for computing its consistent answer (in the classical sense), as there may be atoms which are true or false in the classical consistent answer, but undefined in the answer computed on D_{det} only. This explains in what sense evaluating a query on D_{det} provides, in the general case, an “approximate” evaluation of the classical CQA. Thus, it is worth investigating whether there are classes of constraints and queries for which the classical consistent answer and the answer evaluated on D_{det} coincide. In this regard, in the following we define two classes of full constraints for which the deterministic answer is complete.

Definition 6.9. A set IC of universal constraints is said to be *positive* if every constraint ic in IC contains positive literals only.

Definition 6.10. A set IC of full constraints is said to be *semi-positive* if every constraint in IC contains exactly one negative literal.

It is worth noting that both positive and semi-positive constraints are relevant in practice, as they enable significant classes of constraints to be expressed. For instance, functional dependencies are positive constraints, as well as inclusion dependencies with no existentially quantified variables can be expressed as semi-positive constraints.

We now introduce three lemmas stating properties of repairs in the presence of positive and semi-positive constraints which will be exploited in the following.

Lemma 6.2. *Let D be a two-valued database and IC a set of positive integrity constraints over D . If an atom a is false in D , then there is no repair $R \in \text{repairs}_{\mathcal{PS}}(D, IC)$ such that either $a^+ \in R$ or $a^u \in R$.*

Proof. Assume by contradiction that there is a repair $R \in \text{repairs}_{\mathcal{PS}}(D, IC)$ such that $a^+ \in R$ (resp., $a^u \in R$), and let $R' = R \setminus \{a^+\}$ (resp., $R' = R \setminus \{a^u\}$). Since no $ic \in IC$ contains negative literals, it must be the case that $R' \circ D$ satisfies IC : the constraints containing a are satisfied since a is false in $R' \circ D$, while the other constraints are satisfied since R' coincides with R on the updates performed on the other atoms. This contradicts the minimality of R . \square

Lemma 6.3. *Let D be a two-valued database, a an atom in B_D , and IC a set of positive integrity constraints over D . If there is a repair $R \in \text{repairs}_{\mathcal{PS}}(D, IC)$ such that $a^u \in R$, then there is a constraint $ic \in IC$ containing an occurrence of a such that $D \not\models_{\mathcal{PS}} ic$.*

Proof. We denote the subset of IC consisting of the constraints containing an occurrence of a as IC_a . Assume by contradiction that, for each $ic \in IC_a$, it holds that $D \models_{\mathcal{PS}} ic$. This means that, for each $ic \in IC_a$, at least one atom l_{ic} occurring in ic is false in D . Hence, Lemma 6.2 entails that, for each $ic \in IC_a$, no repair contains an update operation on l_{ic} . This implies that $R' = R \setminus \{a^u\}$ is such that $R' \circ D \models_{\mathcal{PS}} IC$: every $ic \in IC_a$ is satisfied on $R' \circ D$, as ic contains an atom which is false in $R' \circ D$; moreover, the constraints in $IC \setminus IC_a$ are satisfied on $R' \circ D$, since R' coincides with R on the updates performed on the atoms different from a . This contradicts the minimality of R . \square

Lemma 6.4. *Let D be a two-valued database, IC a set of semi-positive constraints over D , $q = (g, \emptyset)$ a query over D , and a an atom in B_D . If a is true (resp., false) in $q(D, IC)$, then a is true (resp., false) in D .*

Proof. We provide the proof for the implication $a \in q(D, IC)^+ \Rightarrow a \in D^+$ only (the other implication can be proved reasoning analogously). Reasoning by contradiction, assume that $a \in q(D, IC)^+$ and $a \in D^-$. Let Up be the database update removing from D every true atom. Since every constraint in IC contains a positive literal, it is easy to see that $Up \circ D \models_{\mathcal{PS}} IC$ holds. This implies that there exists a total repair R for $\langle D, IC \rangle$ consisting of deletions only (R can be found among the subsets of Up). As $a \in D^-$, it follows that $a \in (R \circ D)^-$, which contradicts that $a \in q(D, IC)^+$. \square

The following theorem states that evaluating the deterministic answer of a query whose goal is an extensional predicate is a complete strategy for evaluating its consistent answer (in the classical sense).

Theorem 6.5. *Let D be a two-valued database, IC a set of integrity constraints over D , and $q = (g, \emptyset)$ a Datalog query. Then, $q_{\mathcal{PS}}(D, IC) = q_{det}(D, IC)$ in each of the following cases: (1) IC is positive; (2) IC is semi-positive.*

Proof. Since $q_{det}(D, IC)^+ \subseteq q(D, IC)^+$ and $q_{det}(D, IC)^- \subseteq q(D, IC)^-$ hold (Theorem 6.4), in order to prove the statement it suffices to show that $q_{det}(D, IC)^+ \supseteq q(D, IC)^+$ and $q_{det}(D, IC)^- \supseteq q(D, IC)^-$ hold for both cases (1) and (2). These two properties can be proved analogously, thus we will focus on proving the former for each of the two cases.

It is easy to see that $q_{det}(D, IC)^+ \supseteq q(D, IC)^+$ holds iff there is no atom which is true in $q(D, IC)$ but undefined in D_{det} (no atom in $q(D, IC)^+$ can belong to $q_{det}(D, IC)^-$, as $q_{det}(D, IC)^- \subseteq q(D, IC)^-$ and $q(D, IC)^+ \cap q(D, IC)^- = \emptyset$). Hence, we will prove that $q_{det}(D, IC)^+ \supseteq q(D, IC)^+$ reasoning by contradiction, that is, we assume that there is an atom a which is true in every database resulting from applying a total repair on D and which is undefined in D_{det} (that is, $a^u \in R_{det}$). We consider cases (1) and (2) separately, and show that the existence of a yields a contradiction for both of them.

(1) Lemma 6.3 entails that there is a constraint $ic \in \text{ground}(IC)$ containing an occurrence of a such that $D \not\models_{\mathcal{PS}} ic$. We show that there is a total repair for $\langle D, IC \rangle$ containing a^- . Let S be the set of atoms appearing in ic except a . We build a database update Up as follows:

- $Up := \{a^-\}$;
- for each $ic' \in \text{ground}(IC)$ such that $ic' \neq ic$ and $D \not\models_{\mathcal{PS}} ic'$, we augment Up as follows: $Up := Up \cup \{\beta^-\}$, where β is an atom occurring in ic' s.t. $\beta \notin S$ (the existence of β derives from the fact that $\text{ground}(IC)$ contains no pair of constraints ic_1, ic_2 such that the set of literals occurring in ic_1 is a subset of that of ic_2 : hence, $ic' \neq ic$ implies that ic' contains at least one atom not belonging to S).

That is, Up deletes a and, for each violated constraint ic' different from ic , it makes ic' satisfied by deleting an atom occurring in ic' but not in ic . As Up performs tuple deletions only and IC is positive, applying Up on D cannot result in a database violating some constraint which was satisfied by D . Hence, Up makes D consistent. This implies the existence of a repair R' for $\langle D, IC \rangle$ containing a^- (R' can be found among the subsets of Up , and it must contain a^- as no other atom deletion in Up suffices to make ic satisfied). As R' contains a^- and is total, we obtain that $a \notin q(D, IC)^+$, which is a contradiction.

(2) Without loss of generality, we assume that each constraint in IC contains two literals only (one negative, the other positive). The generalization to the case that constraints contain more than one positive literal is trivial.

Since $a \in q(D, IC)^+$, Lemma 6.4 implies that $a \in D^+$. Consider the database update $R' = R_{det} \setminus \{a^u\}$. Since $a^u \in R_{det}$, it holds that $R' \circ D \not\models_{\mathcal{PS}} IC$ (otherwise, the minimality of R_{det} would be contradicted). Let IC' be the subset of constraints in $\text{ground}(IC)$ violated by $R' \circ D$. It is easy to see that every constraint in IC' is of the form $a, \neg b \supset$ (constraints of the form $\neg a, b \supset$ are satisfied by $R' \circ D$, as $a \in D^+$ and R' does not update a). Specifically, for each $ic : a, \neg b \supset$ in IC' , the facts $R' \circ D \not\models_{\mathcal{PS}} ic$ and $a \in (R' \circ D)^+$ imply that b is either false or undefined in $R' \circ D$. We show that the former cannot hold. Since $a \in q(D, IC)^+$, it must be the case that $b \in q(D, IC)^+$ holds too (ic imposes that a and $\neg b$ cannot be both true). The latter implies that there is no total repair containing b^- , which in turn implies that no partial repair exists in $\text{repairs}_{\mathcal{PS}}(D, IC)$ deleting b (Theorem 6.1). Moreover, the fact that $b \in q(D, IC)^+$ entails also that $b \in D^+$ (Lemma 6.4). Hence, b cannot be false in $R' \circ D$, thus it must be the case that b is made undefined by R' . Therefore, b is an atom in $q(D, IC)^+$ which is made undefined by R_{det} . Thus, we can apply on b the same reasoning applied on a , and expand IC' with the constraints of $\text{ground}(IC)$ which are violated by $R_{det} \setminus \{b^u\}$.

This reasoning can be iterated, until IC' can be expanded no more. That is, at each iteration, a new atom α occurring in constraints of IC' is considered, and IC' is augmented with the constraints in $\text{ground}(IC)$ which are violated

by $R_{det} \setminus \{\alpha^u\} \circ D$. This process ends, as B_D is finite. We denote as A the set of atoms occurring in the constraints of IC' at the end of this process.

Consider the database update $Up = R_{det} \setminus \{\alpha^u \mid \alpha \in A\}$. We show that every $ic : \alpha_1, \neg\alpha_2 \supset$ in $ground(IC)$ is satisfied in $Up \circ D$, reasoning by cases:

- *both α_1 and α_2 belong to A* : for each atom $\alpha \in A$ it holds that $\alpha \in D^+$ (by construction), thus $\alpha \in (Up \circ D)^+$ holds too (as Up does not update atoms in A). Hence, we obtain that $Up \circ D \models_{\mathcal{PS}} ic$, as $\alpha_2 \in A$ and is true in $Up \circ D$
- *neither α_1 nor α_2 belong to A* : $Up \circ D \models_{\mathcal{PS}} ic$ holds since Up coincides with R_{det} on both α_1 and α_2 .
- *exactly one atom among α_1 and α_2 belongs to A* : in this case, it must hold that $ic \notin IC'$ (otherwise, both α_1 and α_2 would belong to A , by construction). We denote the atom among α_1 and α_2 which belongs (resp., does not belong) to A as β (resp., $\bar{\beta}$). The fact that $ic \notin IC'$ and the construction of IC' imply that $(R_{det} \setminus \{\beta^u\}) \circ D \models_{\mathcal{PS}} ic$. As $\beta \in A$ and $\bar{\beta} \notin A$, Up coincides with $R_{det} \setminus \{\beta^u\}$ on both β and $\bar{\beta}$, which implies that $(Up \circ D) \models_{\mathcal{PS}} ic$ holds too.

Hence, we have that $(Up \circ D) \models_{\mathcal{PS}} IC$ and $Up \subset R_{det}$ (the latter containment is proper as A contains at least a and b), which is in contradiction with the minimality of R_{det} . \square

6.5 Computing the Deterministic Repair

In this section we present how the deterministic repair for a database D inconsistent w.r.t. a given set IC of ground constraints can be computed by evaluating a logic program derived from D and IC . Specifically, we derive a (stratified) logic program $Rew(IC)$ such that the (stratified) model of $Rew(IC) \cup D$ defines the deterministic repair for D w.r.t. IC (as a matter of fact, Rew stands for “rewriting”, as it defines a logic program where integrity constraints are suitably re-written into logic rules).

The formal definition of $Rew(IC)$ is given in the following (Definition 6.11). The update atoms which can be derived by the logic program are denoted as $Up(L)$ and $Und(L)$: specifically, given a ground literal $L = a(w)$ (resp., $L = \neg a(w)$), $Up(L)$ will denote the update atom $a(w)^+$ (resp. $a(w)^-$), while $Und(L)$ will denote the update atom $a(w)^u$. The semantics of the other predicates defined by the set of rules of $Rew(IC)$ (such as $MustBeTrue(L)$ and $NotFalse(L)$) will be made clearer in the following.

Definition 6.11. Let IC be a set of ground constraints and ic a (ground) constraint in $sh-ext(IC)$, that is ic is of the form $\bigwedge_{1 \leq i \leq n} L_i \supset L$. We define $rew_1(ic)$ as the following pair of Datalog rules:

$$MustBeTrue(L) \leftarrow \bigwedge_{i=1}^n MustBeTrue(L_i)$$

$$Up(L) \leftarrow MustBeTrue(L), \neg L$$

whereas $Rew_1(IC)$ is equal to $\bigcup_{ic \in sh-ext(IC)} rew_1(ic)$. Moreover $rew_2(ic)$ denotes the following set of Datalog rules:

$$Und(L) \leftarrow \bigwedge_{i=1}^n NotFalse(L_i), \neg L, \neg MustBeTrue(L), \neg MustBeTrue(\neg L)$$

$$NotFalse(L_i) \leftarrow (L_i \wedge \neg Up(\neg L_i)) \vee Up(L_i) \vee Und(L_i) \quad i = 1..n$$

whereas $Rew_2(IC) = \bigcup_{ic \in sh-ext(IC)} rew_2(ic)$. Finally, we define

$$Rew(IC) = Rew_1(IC) \cup Rew_2(IC)$$

The semantics of the predicates and rules introduced in Definition 6.11 can be explained as follows. Intuitively enough, $Rew_1(IC)$ is a program which computes Ψ_{IC} (in the sense that $\Psi_{IC} = \{L \mid MustBeTrue(L) \in \mathcal{MM}(Rew_1(IC))\}$), along with the set of insertion/deletion updates making the source database contain the literals in Ψ_{IC} . Specifically, given a ground literal L , the atom $MustBeTrue(L)$ means that L is *true* in every database which satisfies the constraints. Thus, given a ground integrity constraint ic in $sh-ext(IC)$, the first rule in $rew_1(ic)$ states that if all the literals in the body of ic are true in every database consistent w.r.t. IC , then, in order to satisfy ic , the head literal must be true too. The second rule properly defines (insertion/deletion) update actions which should be accomplished on the source database, as it states that if a literal L is false in the source database while it should be *true* to satisfy the constraints, then an update action making L *true* must be performed.

As regards $Rew_2(IC)$, it computes the set of update operations assigning *undefined* to database atoms. Specifically, given a ground literal L , the atom $NotFalse(L)$ means that applying the set of update operations computed by the program does not result in a database where L is false. Hence, given a ground integrity constraint ic , the first rule in $rew_2(ic)$ has the following meaning: if performing the set of update operations computed by the program makes no body literal of ic false, but its head literal L is false in the source database, then an update action making L undefined must be performed, provided that it has not been derived that L must be either true or false by $Rew_1(IC)$.

In the following, given a ground literal $L = a(w)$ (resp., $L = \neg a(w)$), we denote the atom $MustBeTrue(L)$ as $a(w)^t$ (resp. $a(w)^f$), and the atom $NotFalse(L)$ as $a(w)^{tu}$ (resp. $a(w)^{fu}$), where the superscripts *tu* and *fu* stand for “*true or undefined*”, and *false or undefined*”, respectively.

Example 6.7. Consider the following set IC of (ground) integrity constraints

$$\begin{aligned} & \mathbf{a} \supset \\ & \neg \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c} \supset \end{aligned}$$

The set $sh-ext(IC)$ of constraints is as follows

$$\begin{aligned} ic_1 &: \supset \neg a \\ ic_2 &: \neg a \wedge b \supset \neg c \\ ic_3 &: \neg a \wedge c \supset \neg b \\ ic_4 &: b \wedge c \supset a \end{aligned}$$

The program $Rew_1(IC)$ is

$$\begin{aligned} rew_1(ic_1) &: \{ a^f \leftarrow, a^- \leftarrow a^f, a \} \\ rew_1(ic_2) &: \{ c^f \leftarrow a^f \wedge b^t, c^- \leftarrow c^f, c \} \\ rew_1(ic_3) &: \{ b^f \leftarrow a^f \wedge c^t, b^- \leftarrow b^f, b \} \\ rew_1(ic_4) &: \{ a^t \leftarrow b^t \wedge c^t, a^+ \leftarrow a^t, \neg a \} \end{aligned}$$

whereas $Rew_2(IC)$ is

$$\begin{aligned} a^u &\leftarrow a, \neg a^t, \neg a^f \\ c^u &\leftarrow a^{fu}, b^{tu}, c, \neg c^t, \neg c^f \\ b^u &\leftarrow a^{fu}, c^{tu}, b, \neg b^t, \neg b^f \\ a^u &\leftarrow b^{tu}, c^{tu}, \neg a, \neg a^t, \neg a^f \\ \\ a^{fu} &\leftarrow (\neg a \wedge \neg a^t) \vee a^f \vee a^u \\ b^{tu} &\leftarrow (b \wedge \neg b^f) \vee b^t \vee b^u \\ c^{tu} &\leftarrow (c \wedge \neg c^f) \vee c^t \vee c^u \end{aligned}$$

The program $Rew(IC)$ contains standard rules with negation and denial rules (rules with empty heads). The set of stable models for $Rew(IC) \cup \mathcal{DB}$ consists of all the stable models of $Rew_1(IC) \cup Rew_2(IC) \cup D$. As regards the rules in $Rew_1(IC) \cup Rew_2(IC)$, observe that rules in $Rew_1(IC)$ are semi-positive (they depend through negation only on database atoms), whereas atoms defined in $Rew_2(IC)$ depend through negation only on atoms defined in $Rew_1(IC)$. Therefore, the program $Rew_1(IC) \cup Rew_2(IC)$ is stratified and, consequently, $Rew(IC) \cup \mathcal{DB}$ has a unique stable model which is the stratified model of $Rew_1(IC) \cup Rew_2(IC) \cup D$.

Let D be a two-valued database, IC be a set of integrity constraints on D and M the stratified model for $Rew(ground(IC)) \cup D$. Then, $S(M) = \langle \{a(t) \mid a(t)^+ \in M\}, \{a(t) \mid a(t)^u \in M\} \{a(t) \mid a(t)^- \in M\} \rangle$ denotes the database update derived from M .

Theorem 6.6. *Let D be a two-valued database and IC a set of full constraints on D . Let M be the unique model in $\mathcal{SM}(Rew(ground(IC)) \cup D)$. Then, the database update $S(M)$ is the deterministic repair for $\langle D, IC \rangle$.*

Proof. Let $P_1 = Rew_1(ground(IC))$. We denote the subset of rules of P_1 whose heads consist of update atoms as $P_{1,2}$, and the complementary subset of rules as $P_{1,1}$. Let $N_1 = T_{P_{1,2}}(T_{P_{1,1}}^\infty(\emptyset) \cup D)$. By construction, it follows that $N_1 = \{Up(A) \mid A \in \Psi_{IC}\}$. Let $P_2 = Rew_2(ground(IC))$. We denote as

$P_{2,1}$ the subset of P_2 defining update atoms (i.e., atoms with adornment \mathbf{u}), whereas $P_{2,2} = P_2 - P_{2,1}$ denotes the set of remaining rules (i.e., rules defining atoms with adornment \mathbf{tu} or \mathbf{fu}). Let $N_2 = T_{\mathcal{P}_{2,1}}^\infty(T_{(\mathcal{P}_2 \cup \mathcal{D}\mathcal{B} \cup N_1)}^\infty(\emptyset))$, i.e., N_2 is $S(M)^u$. To prove the statement, it suffices to show that $N_2 = \{A^u \mid A^u \in R_{det}\}$ (in fact, it is easy to see that R_{det} and $S(M)$ contain the same set of update atoms assigning true or false, as this set must be equal to $\Psi_{ground(IC)}$).

For each $a^u \in R_{det}$, there is at least a ground constraint r in $Red(ground(IC))$ of the form $r = \alpha \wedge a \wedge b \supset$, where α is a conjunction of ground literals, such that $b^u \in R_{det}$ (without loss of generality, we assume that both a and b are positive literals in r). We now show that the program P_2 derives the atoms a^u and b^u , exploiting the fact that $Up(-a)$ and $Up(-b)$ have not been derived in the previous phase. In fact, considering the rules in P_2 obtained from constraint r , we have that, since the extended version contains both constraints $\alpha \wedge a \supset \neg b$ and $\alpha \wedge b \supset \neg a$, the following dependencies¹ $a^u \leftarrow b^{tu} \leftarrow (b \wedge \neg b^f)$ and $b^u \leftarrow a^{tu} \leftarrow (a \wedge \neg a^f)$ must hold. Since $D \models a \wedge b$ and $N_1 \not\models a^f \wedge b^f$, both a^u and b^u are derived. The same reasoning can be applied for the cases where one or both literals appear negated in r . Therefore all atoms which are undefined in R_{det} are also derived undefined in M .

Vice versa, it cannot be the case that $S(M)$ contains an update atom which is not in R_{det}^u . In fact, it is easy to see that R_{det} corresponds to a model of $Rew(ground(IC)) \cup D$, thus $R_{det}^u - S(M)^u \neq \emptyset$ would contradict the minimality of M , since $R_{det}^+ = S(M)^+$ and $R_{det}^- = S(M)^-$. \square

Example 6.8. Consider the database D which consists of $D^+ = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ and the set IC of integrity constraints of Example 6.7. Then, $S(\mathcal{SM}(Rew(IC) \cup D)) = \langle \{\}, \{\mathbf{b}, \mathbf{c}\}, \{\mathbf{a}\} \rangle$ which is the deterministic repair for $\langle D, IC \rangle$.

Corollary 6.2. *Let D be a two-valued database and IC a set of integrity constraints on D . The deterministic repair for $\langle D, IC \rangle$ can be computed in polynomial time in the size of D .*

Proof. The computation of the deterministic repair of $\langle D, IC \rangle$ can be carried out by evaluating the stable model of $Rew(ground(IC)) \cup D$. As $Rew(ground(IC))$ is a stratified program and the computation of the stable model of stratified programs can be computed in polynomial time, we have that the deterministic repair can be computed in polynomial time as well. \square

Corollary 6.3. *Let D be a two-valued database, IC a set of integrity constraints and $q = (g, P)$ a Datalog query. Then, $q_{det}(D, IC)$ can be computed in polynomial time.*

¹ A dependency from B to A , denoted by $A \leftarrow B$, means that there is rule with A in the head and B in the body.

Proof. The answer of q over D_{det} can be obtained by considering the well-founded semantics of the program $P \cup P^u \cup D_{det}^+$, where P^u is the set of rules $\{a(w) \leftarrow \neg a(w) \mid a(w) \in D_{det}^u\}$ (rules in P^u enable the atoms in D_{det}^u to be derived as *undefined*). From Corollary 6.2 (which states that D_{det} can be computed in polynomial time) and the fact that evaluating the well-founded model of a program can be accomplished in polynomial time, we obtain that $q_{det}(D, IC)$ can be computed in polynomial time too. \square

In the proof of Corollary 6.3, we proved that $q_{det}(D, IC)$ is polynomial-time computable exploiting the fact that it can be obtained from the well-founded semantics of the program P on D_{det} (where P is the program specified in q). Since it is well-known that the computation of the well-founded model is generally more expensive than the computation of a stratified model (although both of them can be accomplished in polynomial time), the evaluation of the deterministic answer of a query would be likely to be carried out more efficiently if it were possible to evaluate it in terms of the stratified model (rather than the well-founded model) of a logic program. In this regard, given a query $q = (g, P)$ over a database D inconsistent w.r.t. a set of integrity constraints IC , we now show how P can be rewritten into a set of logic rules $Rew(P)$ such that the stratified model of the logic program $Rew(ground(IC)) \cup Rew(P) \cup D$ defines the deterministic answer of q .

Let (g, P) be a query (where P is a Datalog program and g a predicate defined in P), and r a rule in P of the form:

$$p_0(X) \leftarrow \bigwedge_{i=1}^m q_i(X_i), \bigwedge_{j=1}^n p_j(X_j), \Phi$$

where each p_j ($0 \leq j \leq n$) is a derived predicate symbol, q_i ($1 \leq i \leq m$) is a base predicate symbol, and Φ is a conjunction of built-in atoms. We denote as $rew^T(r)$ the rule:

$$p_0^T(X) \leftarrow \bigwedge_{i=1}^h \left(\left(q_i(X_i) \wedge \neg q_i(X_i)^- \wedge \neg q_i(X_i)^u \right) \vee q_i(X_i)^+ \right), \\ \bigwedge_{j=1}^n p_j^T(X_j), \Phi$$

which defines when $p_0(X)$ can be derived as *true*. Specifically, the definition of p_0^T states that $p_0(X)$ can be derived as *true* if the body of r evaluates to true on D_{det} . In order to determine this, the value of each $q_i(X_i)$ w.r.t. D_{det} is evaluated by taking into account the effects of applying the updates of the deterministic repair on D : $q_i(X_i)$ is true in D_{det} if either it must be inserted into the database according to R_{det} (that is, R_{det} contains $q_i(X_i)^+$), or it is true in D and R_{det} does not change its truth value (that is, both $q_i(X_i)^-$ and $q_i(X_i)^u$ are false in R_{det}).

Moreover, we denote as $rew^U(r)$ the rule:

$$p_0^U(X) \leftarrow \bigwedge_{i=1}^h \left(\left(q_i(X_i) \wedge \neg q_i(X_i)^- \right) \vee q_i(X_i)^+ \vee q_i(X_i)^u \right), \\ \bigwedge_{j=1}^n \left(p_j^U(X_j) \vee p_j^T(X_j) \right), \neg p_0^T(X), \Phi$$

which defines when $p_0(X)$ can be derived as *undefined*, by taking into account the update operations of R_{det} .

Moreover, we define

$$Rew^T(P) = \{rew(r) \mid r \in P\} \text{ and } Rew^U(P) = \{rew^U(r) \mid r \in P\},$$

and

$$Rew(P) = Rew^T(P) \cup Rew^U(P).$$

The evaluation of the stratified model of $Rew(ground(IC)) \cup Rew(P)$ over D is equivalent to evaluating $Rew(P)$ over $\mathcal{SM}(Rew(ground(IC)) \cup D)$, as the predicates defined in $Rew(P)$ depend on the predicates of both $Rew(ground(IC))$ and D . Basically, the stratified model of $Rew(ground(IC))$ defines the deterministic repair, thus evaluating the stratified model of $Rew(P)$ on $\mathcal{SM}(Rew(ground(IC)) \cup D)$ corresponds to taking into account the update operations performed by the deterministic repair to evaluate the deterministic answer.

It is easy to see that, given a query $q = (g, P)$, the semantics of $Rew(P)$ defines the deterministic answer of q in the following sense: a g -atom is true (resp., undefined) iff g^T (resp., g^U) belongs to the stratified model of $Rew(ground(IC)) \cup Rew(P) \cup D$, and it is false iff the latter contains neither g^T nor g^U .

6.6 Discussion

We have presented a logic framework for repairing and querying inconsistent databases which allows us to compute “approximate” consistent answers in polynomial time. We have considered three valued databases, i.e. databases whose atoms can be either *true* or *undefined* (whereas missing atoms are *false*). We have proposed a three-valued semantics for constraint satisfaction, called *partial*, and presented the notions of repair and consistent query answer under this semantics. We have shown that in querying possibly inconsistent databases under the proposed semantics the answer is safe (*true* and *false* atoms in the answers are, respectively, *true* and *false* in the classical CQA) and can be computed in polynomial time. We have also shown that deterministic repairs can be computed by means of logic programs derived from constraints, which can be evaluated by means of standard systems, such as *DLV*, *Smodels* and *XSB* [72, 59, 69].

The management of inconsistent databases has been investigated in several works from a different standpoint, that is measuring the degree of inconsistency of data. Most of these works exploit different forms of paraconsistent reasoning. In [45], a framework for measuring the inconsistency of a knowledgebase was presented, where knowledgebases are represented as first-order

logic formulas and *quasi-classical* (QC) logic is used to support paraconsistent reasoning. The latter work can be viewed as an extension of previous QC-logic based approaches for measuring inconsistencies in propositional theories [51, 52] to the first-order case. In the latter approaches, inconsistent sets of formulae are analyzed in terms of their quasi-classical models, and the degree of inconsistency is measured by suitably measuring the inconsistency inside these models.

Measures of information and contradiction were reviewed in [53]. The paper discusses some dimensions for measuring inconsistent information (that is, different ways of describing inconsistency) and several approaches to measuring inconsistent information. These approaches are based on different ways of weighting conflicts: some of them assume that the importance of a conflict depends on the number of formulae of the knowledgebases implied in the contradiction, while others measure the importance of a conflict in terms of number of atoms on which we have contradictory information.

A general framework to resolve contradictions in first order logic systems was proposed in [64], where the *semantics of weighted mc-subsets* was introduced as a way of reasoning in inconsistent systems. This semantics enables reconciling contradictions and deriving plausible beliefs about any statement including ambiguous ones.

In [65], an axiomatic approach for measuring inconsistency in databases was proposed, in the presence of functional dependencies. In this paper, several properties were introduced (in terms of axioms) for deciding whether an inconsistency metric is reasonable, and both old and new metrics were compared with respect to these axioms.

Approximate Probabilistic Query Answering over Inconsistent Databases

In this chapter we present a framework for querying inconsistent databases in the presence of (particular sets of) functional dependencies. Both the notions of repair and query answer differ from the classical ones. Specifically, databases are repaired by means of tuple updates whereas query answers are *probabilistic answers*, that is tuples associated with probabilities (the probability associated with a tuple depends on the number of repaired databases from which the tuple can be derived by evaluating the given query). In the classical framework of consistent query answering, in the presence of functional dependencies, tuple deletions are the only operations that can be performed in order to restore the consistency of an inconsistent database. However, deleting a tuple to remove an integrity violation potentially eliminates useful information in that tuple. The proposed repairing strategy copes with this problem by performing tuple updates, so that the information in the source database is preserved better. A drawback of the notion of consistent query answer is that it doesn't allow us to discriminate among "undefined" tuples, namely tuples which can be derived from a proper non empty subset of the repaired databases. In order to cope with this problem and obtain more informative query answers, we propose the notion of probabilistic query answers. We present a technique for computing probabilistic query answers. Such a technique allows us to compute *approximate* probabilistic query answers in polynomial time.

7.1 Introduction

Most of the works on repairing inconsistent databases do not take into account update operations as primitive for restoring consistency (they just consider delete and insert operations). However, deleting a tuple to remove an integrity violation potentially eliminates useful information in that tuple. For instance, consider the following example.

Example 7.1. Consider the relation schema $affiliation(EMP, Dept, City)$ with the functional dependency $fd : Dept \rightarrow City$, stating that a department is located in a unique city. Consider now the following inconsistent relation $affiliation$:

$$affiliation$$

<i>Emp</i>	<i>Dept</i>	<i>City</i>
<i>john</i>	<i>cs</i>	<i>rome</i>
<i>bob</i>	<i>cs</i>	<i>milan</i>

According to most of the approaches proposed so far, the above database can be repaired by means of tuple deletions, thus the following repaired databases can be obtained:

$$affiliation_1 = \{ affiliation(john, cs, rome) \}$$

$$affiliation_2 = \{ affiliation(bob, cs, milan) \}$$

Consider the query $q = \pi_{EMP}(\sigma_{Dept='cs'}affiliation)$ asking for the employees of the department cs . The consistent query answer gives the employees $john$ and bob as undefined, as $john$ works for cs according to the first repaired database only, whereas bob works for cs according to the second repaired database only.

In the previous example, if we suppose that each tuple has an “unreliable” value only on the attribute $City$ and the remaining values are “reliable”, then we would expect that the previous query gives both $john$ and bob as certain, as the query does not regard the attribute $City$. Indeed, repairing inconsistent databases by means of tuple deletion eliminates useful information present in deleted tuples, e.g. information which is not involved in a constraint violation.

In order to cope with this problem, we propose a framework wherein inconsistent databases are repaired by means of tuple updates, as shown in the following example.

Example 7.2. Consider the inconsistent database $affiliation$ of Example 7.1. It can be repaired by assigning the same value on the attribute $City$ to each tuple, that is we assign a unique city to the department cs . This value can be either $rome$ or $milan$, as these values come from the source database. Thus, there exist two repaired databases, namely:

$$affiliation_1 = \{ affiliation(john, cs, rome), affiliation(bob, cs, rome) \}$$

$$affiliation_2 = \{ affiliation(john, cs, milan), affiliation(bob, cs, milan) \}$$

Consider the query q of Example 7.1. By assuming the above repaired databases, q gives the employees $john$ and bob as certain, as they are derivable from all the repaired databases.

The technique we propose allows us to compute certain (i.e. tuples derivable from all or from none of the repaired databases) and uncertain answers

(i.e. tuples derivable from a proper not empty subset of the repaired databases). Moreover, each tuple in the answer is associated with a probability, which depends on the number of repaired databases from which the tuple can be derived (although the framework can be easily adapted so that the probabilities are determined by other criteria).

Example 7.3. Consider the following relation *emp*:

<i>emp</i>	
<i>Name</i>	<i>Dept</i>
<i>john</i>	<i>cs</i>
<i>john</i>	<i>math</i>
<i>bob</i>	<i>cs</i>
<i>bob</i>	<i>physics</i>

which is inconsistent w.r.t. the functional dependency $fd : Name \rightarrow Dept$ and the query $q = \pi_{Dept}(emp)$ asking for the departments of the employees. The intuition suggests that *cs* should be the most probable department as each employee could work for it, whereas *math* and *physics* should be less probable as only *john* could work for the former and only *bob* could work for the latter. The probabilistic answer gives $\{(cs, 3/4), (math, 2/4), (physics, 2/4)\}$ according to the previous consideration. Observe that, under the standard notion of consistent query answer, the departments *cs*, *math* and *physics* are undefined and there is no discrimination among them.

Thus, the proposed approach allow us to exploit better information in the source inconsistent database in two main aspects: (i) repairing by means of tuple updates, which is more fine-grained than the approaches based on tuple deletions, allows us to preserve useful information in the source database; (ii) probabilistic query answering allow us to discriminate among undefined tuples.

The chapter is organized as follows. In Section 7.2, the probabilistic relational model presented in [37] is introduced. Section 7.3 presents a definition of repaired databases and a “condensed” form to represent the set of all repaired databases. Section 7.4 presents the notion of probabilistic query answer and shows how to compute an approximation of it in polynomial time. Finally, in Section 7.5 related works are discussed and conclusions are drawn.

7.2 Probabilistic Relational Model

We recall the probabilistic relational model presented in [37] (see also [31, 33]). A *probabilistic relation* corresponds to an ordinary relation where the membership of a single tuple in the relation is affected by a probabilistic event. We distinguish between *basic* and *complex* events. Tuples of base relations are associated with basic events. Special events are the certain event \top , which is

associated with deterministic tuples, and the impossible event \perp , which is associated with tuples that are not in the database. Each basic event e is associated with a (fixed) probability value denoted by $p(e)$; the probability of \top is 1, whereas the probability of \perp is 0. When new relations are derived by means of Probabilistic Relational Algebra (PRA) operators, each tuple in a derived relation depends on the tuples of the argument relation(s) from which it was derived. In order to express this relationship, we use complex events, which are Boolean combinations of events. Starting from the probabilities for the basic events, the probabilities of complex events can be computed by means of a function \mathcal{P} . The probability associated with a general event e is denoted by $Pr(e)$ and is equal to $p(e)$ if e is a basic event, whereas is equal to $\mathcal{P}(e)$ if e is a complex event. Observe that the function \mathcal{P} takes into account the dependencies among basic events.

This model is based on an *intensional semantics*; this means that each tuple of a relation is associated with an event expression and the PRA operators also manipulate these expressions. The issue of associating probabilities with these expressions is dealt with separately. A probabilistic relational model based on an *extensional semantics* was proposed in [36]. In this model probabilities are attached to tuples; when applying an operator of the relational algebra, the probabilities of the result tuples are computed as a function of the tuple probabilities in the argument relation(s). This approach doesn't always work.

A probabilistic tuple t^p on a relation schema $R(W)$ is a pair $\langle t, e \rangle$, where t is a tuple over $R(W)$ and e is an event. A probabilistic relation on a relation schema $R(W)$ is a set of probabilistic tuples $t^p = \langle t, e \rangle$ such that t is defined over $R(W)$. A probabilistic database D^p is a set of probabilistic relations plus a probabilistic function Pr . In the following, for a given probabilistic tuple t^p , t denotes the corresponding standard tuple; analogously, r and D denote the (standard) relation and database corresponding to the probabilistic relation r^p and the probabilistic database D^p , respectively.

The PRA operators are defined as follows.

- *Selection.* Let r^p be a probabilistic relation

$$\sigma_\theta(r^p) = \{\langle t, e \rangle \mid \langle t, e \rangle \in r^p \wedge t \in \sigma_\theta(r)\}$$

- *Projection.* Let r^p be a probabilistic relation over $R(W)$ and A be a subset of W

$$\pi_A(r^p) = \{\langle t, e \rangle \mid t \in \pi_A r \wedge e = \bigvee_{\langle t', e' \rangle \in r^p \wedge t'[A]=t} e'\}$$

- *Cartesian product.* Let r^p and s^p be probabilistic relations

$$r^p \times s^p = \{\langle t_r.t_s, e_r \wedge e_s \rangle \mid \langle t_r, e_r \rangle \in r^p \wedge \langle t_s, e_s \rangle \in s^p\}$$

- *Union.* Let r^p be a probabilistic relation over $R(W)$ and s^p be a probabilistic relation over $S(W)$

$$r^p \cup s^p = \{\langle t, e \rangle \mid t \in r \cup s \wedge e = \bigvee_{\langle t, e' \rangle \in r^p \vee \langle t, e' \rangle \in s^p} e'\}$$

- *Difference.* Let r^p be a probabilistic relation over $R(W)$ and s^p be a probabilistic relation over $S(W)$

$$r^p - s^p = \{\langle t, e \rangle \mid \langle t, e \rangle \in r^p \wedge \nexists \langle t, e' \rangle \in s^p\} \cup \{\langle t, e_r \wedge \neg e_s \rangle \mid \langle t, e_r \rangle \in r^p \wedge \langle t, e_s \rangle \in s^p\}$$

Observe that the operators $\sigma, \pi, \times, \cup$ and $-$ are overloaded as we have used the same operators of standard relational algebra.

Example 7.4. Consider the probabilistic database D^p consisting of the following probabilistic relations emp and $dept$:

<i>emp</i>		
<i>EName</i>	<i>Dept</i>	
<i>john</i>	<i>cs</i>	e_1
<i>john</i>	<i>math</i>	e_2

<i>dept</i>		
<i>DName</i>	<i>City</i>	
<i>cs</i>	<i>rome</i>	d_1
<i>math</i>	<i>rome</i>	d_2

Consider now the query:

$$q = \pi_{City}(\sigma_{EName='john' \wedge Dept=DName}(emp \times dept))$$

asking for the cities where *john* works. In the evaluation of q , firstly the cartesian products $emp \times dept$ is computed, giving the result below:

<i>EName</i>	<i>Dept</i>	<i>DName</i>	<i>City</i>	
<i>john</i>	<i>cs</i>	<i>cs</i>	<i>rome</i>	$e_1 \wedge d_1$
<i>john</i>	<i>cs</i>	<i>math</i>	<i>rome</i>	$e_1 \wedge d_2$
<i>john</i>	<i>math</i>	<i>cs</i>	<i>rome</i>	$e_2 \wedge d_1$
<i>john</i>	<i>math</i>	<i>math</i>	<i>rome</i>	$e_2 \wedge d_2$

Next, the selection operation $\sigma_{EName='john' \wedge Dept=DName}(emp \times dept)$ is computed and the following result is obtained:

<i>EName</i>	<i>Dept</i>	<i>DName</i>	<i>City</i>	
<i>john</i>	<i>cs</i>	<i>cs</i>	<i>rome</i>	$e_1 \wedge d_1$
<i>john</i>	<i>math</i>	<i>math</i>	<i>rome</i>	$e_2 \wedge d_2$

Finally, the projection operation $\pi_{City}(\sigma_{EName='john' \wedge Dept=DName}(emp \times dept))$ is computed, giving the result:

<i>City</i>	
<i>rome</i>	$(e_1 \wedge d_1) \vee (e_2 \wedge d_2)$

Thus, the answer contains *rome*, whose associated event is $(e_1 \wedge d_1) \vee (e_2 \wedge d_2)$; this means that *john* works in *rome* if either (i) he works for the department *cs* (event e_1) and *cs* is located in *rome* (event d_1), or (ii) he works for the department *math* (event e_2) and *math* is located in *rome* (event d_2).

We point out that, given a probabilistic database D^p and a query q , the probability associated with the tuples in $q(D^p)$ is computed by means of the function Pr which takes into account the relations among basic events.

Example 7.5. Consider the probabilistic database D^p and the query q of Example 7.4. Given the probabilities for the basic events e_1, e_2, d_1, d_2 and assuming that all events are mutually independent $q(D^p)$ gives the tuple *rome* along with its probability, namely $Pr((e_1 \wedge d_1) \vee (e_2 \wedge d_2)) = Pr(e_1) \times Pr(d_1) + Pr(e_2) \times Pr(d_2) - Pr(e_1) \times Pr(d_1) \times Pr(e_2) \times Pr(d_2)$.

7.3 Repairing

In this chapter, a minimal set of updates which leads a database to a consistent state will be called *repair* (it will be precisely defined in the following); a *repaired database* is the result of applying a repair over the original database. In order to distinguish the proposed notion of repaired database from the classical one, we will refer to the latter as id-repaired database¹. Thus, given a database D and a set of integrity constraints IC , D' is an id-repaired database derived from D if $D' \models IC$ and the pair $(D' - D, D - D')$ is minimal under set inclusion, i.e. there is no database $D'' \neq D'$ such that $D'' \models IC$ and both containments $D'' - D \subseteq D' - D$ and $D - D'' \subseteq D - D'$ hold. Given a database D and a set IC of integrity constraints, $repaired_{id}(D, IC)$ denotes the set of all the possible id-repaired databases for $\langle D, IC \rangle$.

We assume two disjoint, infinite sets **dom** and **var** of *constants* and *variables* respectively. A *symbol* is either a constant or a variable.

A *condensed tuple* ct over a relation schema $R(W)$, where W is a set of attributes, is a total mapping from W to **dom** \cup **var**; a *condensed relation* over $R(W)$ is a set of condensed tuples over the same schema $R(W)$, whereas a *condensed database* is a set of condensed relations. Each variable V in a condensed database has a domain $dom(V) \subseteq \mathbf{dom}$ of possible values. The value of ct on an attribute A in W is denoted $ct(A)$; this is extended so that for $Z \subseteq W$, $ct[Z]$ denotes the condensed tuple z over Z such that $ct(A) = z(A)$ for each $A \in Z$. The set of variables in ct is denoted by $var(ct)$, whereas the set of constants in ct is denoted by $const(ct)$. Analogously, for a given relation r (resp. database D), $var(r)$ and $const(r)$ (resp. $var(D)$ and $const(D)$) denote respectively the sets of variables and constants in r (resp. D). Moreover, ct (resp. r , D) is said to be *ground* if $var(ct) = \emptyset$ (resp. $var(r) = \emptyset$, $var(D) = \emptyset$). Ground condensed tuples (resp. relations, databases) are also called simply tuples (resp. relations, databases).

A (*ground*) *substitution* for a set of variables $\{V_1, \dots, V_k\}$, $k \geq 0$, is a set of pairs $\{V_1/c_1, \dots, V_k/c_k\}$ where c_1, \dots, c_k are constants such that $c_i \in dom(V_i)$ for $i = 1..k$. We also use the notation $\theta[1] = \{V \mid V/c \in \theta\}$ and

¹ id-repaired database stands for repaired database obtained by means of tuple insertions and deletions.

$\theta[2] = \{c \mid V/c \in \theta\}$ to denote the sets of variables and constants in θ , respectively. $\theta(V) = c$ if there is a pair $V/c \in \theta$, otherwise $\theta(V) = V$.

The application of a substitution θ to a condensed tuple $ct = \langle p_1, \dots, p_n \rangle$ is $\theta(ct) = \langle \theta(p_1), \dots, \theta(p_n) \rangle$. Analogously, the application of a substitution θ to a condensed relation cr is $\theta(cr) = \{\theta(ct) \mid ct \in cr\}$, whereas the application of θ to a condensed database D_c is $\theta(D_c) = \{\theta(cr) \mid cr \in D_c\}$.

Given a condensed relation cr (resp. database D_c), $\mathbf{G}(cr)$ (resp. $\mathbf{G}(D_c)$) denotes the set of all the (ground) relations (resp. databases) that can be obtained from cr (resp. D_c) by replacing all the variables in cr (resp. D_c) with constants belonging to the domains associated with variables.

Definition 7.1. *Canonical functional dependencies.* Let $R(W)$ be a relation schema and FD be a set of functional dependencies in standard form² over $R(W)$. FD is said to be in *canonical form* if $\forall X \rightarrow A \in FD$ does not exist a functional dependency $Y \rightarrow B \in FD$ such that $A \in Y$.

In the rest of the chapter, we consider sets of functional dependencies in canonical form.

Let $R(W)$ be a relation schema, FD be a set of functional dependencies over $R(W)$ and r be an instance of $R(W)$. An *update operation* for r is a pair $u = (t, t')$ of tuples over $R(W)$ s.t. $t \in r \wedge t \neq t'$. The intuitive meaning of $u = (t, t')$ is that t is replaced by t' , i.e. the updated relation obtained from r by applying u is $u(r) = r - \{t\} \cup \{t'\}$. Given a set of update operations Up , then we define the sets $Up^- = \{t \mid \exists (t, t') \in Up\}$ and $Up^+ = \{t' \mid \exists (t, t') \in Up\}$. We say that Up is *coherent* if it does not contain two distinct update operations (t, t') and (t_1, t_2) such that either $t = t_1$ or $t = t_2$, that is (i) the same tuple t cannot be replaced by two distinct tuples t' and t_2 , and (ii) a tuple t which is replaced by a tuple t' cannot be used to replace in turn a tuple t_1 .

Given a set Up of update operations for r , we denote by $Up(r)$ the updated relation obtained from r by applying all the update operations in Up , i.e. $Up(r) = r - Up^- \cup Up^+$. Moreover, we define the set $update(Up) = \{(t, A) \mid \exists (t, t') \in Up, A \in W \text{ s.t. } t(A) \neq t'(A)\}$. If a pair (t, A) is in $update(Up)$, then we say that Up *modifies the value of the tuple t on the attribute A* .

We say that the value of a tuple $t \in r$ on an attribute $A \in W$ is *uncertain* if there exists a tuple $t' \in r$ and a functional dependency $fd: X \rightarrow A \in FD$ such that $\{t, t'\} \not\models fd$, that is $t[X] = t'[X]$ and $t(A) \neq t'(A)$. A set Up of update operations for r is said to be *feasible* (w.r.t. r) if it modifies only uncertain values.

Definition 7.2. *Repair and repaired database.* Let $R(W)$ be a relation schema, FD be a set of functional dependencies over $R(W)$ and r be an instance of

² We consider functional dependencies of the form $X \rightarrow A$, where X is a set of attributes whereas A is an attribute.

$R(W)$. A repair for $\langle r, FD \rangle$ is a coherent and feasible set Up of update operations for r such that (i) $Up(r) \models FD$ and (ii) there is no set of update operations Up' such that $update(Up') \subset update(Up) \wedge Up'(r) \models FD$. The set of all the possible repaired relations for $\langle r, FD \rangle$ is denoted as $repaired_U(r, FD)$.

Thus, a repair is a minimal set of attribute value modifications which makes a database consistent by modifying only uncertain values. Repaired relations are consistent relations derived from the source relation by means of repairs.

Given an inconsistent database D and a set FD of functional dependencies, a repaired database is obtained by repairing each inconsistent relation in D . We denote by $repaired_U(D, FD)$ the set of all the repaired databases for $\langle D, FD \rangle$.

Example 7.6. Consider the relation schema $emp(Name, Dept, City)$ with the functional dependencies $FD = \{Name \rightarrow City, Dept \rightarrow City\}$. Consider now the following inconsistent relation r :

<i>emp</i>		
<i>Name</i>	<i>Dept</i>	<i>City</i>
<i>john</i>	<i>math</i>	<i>milan</i>
<i>john</i>	<i>cs</i>	<i>rome</i>
<i>bob</i>	<i>cs</i>	<i>venice</i>
<i>mary</i>	<i>physics</i>	<i>naples</i>

Intuitively, in order to make the relation consistent the first three tuples should have the same value on the attribute *City*. There are three possible repairs for $\langle r, FD \rangle$:

$$\begin{aligned}
 Up_1 &= \{ (emp(john, cs, rome), \quad emp(john, cs, milan) \quad), \\
 &\quad (emp(bob, cs, venice), \quad emp(bob, cs, milan) \quad) \} \\
 Up_2 &= \{ (emp(john, math, milan), emp(john, math, rome) \quad), \\
 &\quad (emp(bob, cs, venice), \quad emp(bob, cs, rome) \quad) \} \\
 Up_3 &= \{ (emp(john, math, milan), emp(john, math, venice) \quad), \\
 &\quad (emp(john, cs, rome), \quad emp(john, cs, venice) \quad) \}
 \end{aligned}$$

By applying the above repairs on r , the following repaired relations are obtained:

$$\begin{aligned}
 r_1 &= \{ emp(john, math, milan), emp(john, cs, milan), \\
 &\quad emp(bob, cs, milan), \quad emp(mary, physics, naples) \} \\
 r_2 &= \{ emp(john, math, rome), emp(john, cs, rome), \\
 &\quad emp(bob, cs, rome), \quad emp(mary, physics, naples) \} \\
 r_3 &= \{ emp(john, math, venice), emp(john, cs, venice), \\
 &\quad emp(bob, cs, venice), \quad emp(mary, physics, naples) \}
 \end{aligned}$$

Therefore $repaired_U(r, FD) = \{r_1, r_2, r_3\}$.

It is worth noting that, in the previous example, each repair updates the value of the attribute *City* so that all conflicting tuples have the same value for this attribute. The minimality guarantees that only values appearing in conflicting tuples are used. For instance, the following set of update actions

$$\mathcal{U} = \{ (emp(john, math, milan), emp(john, math, naples)), \\ (emp(john, cs, rome), emp(john, cs, naples)), \\ (emp(bob, cs, venice), emp(bob, cs, naples)) \}$$

makes the database consistent, but as it is not minimal, it is not a repair.

Definition 7.3. *Condensed representation.* Let $R(W)$ be a relation schema, FD be a set of functional dependencies over $R(W)$ and r be an instance of $R(W)$. A condensed representation of all the repaired relations derivable from $\langle r, FD \rangle$, denoted $r_{\mathcal{F}D}$, is a condensed relation s.t. $\mathbf{G}(r_{\mathcal{F}D}) = repaired_U(r, FD)$ and $var(r_{\mathcal{F}D})$ is minimal modulo renaming of variables.

$D_{\mathcal{F}D}$ denotes the condensed database “representing” all the repaired databases derivable from $\langle D, FD \rangle$, i.e. the condensed database such that $\mathbf{G}(D_{\mathcal{F}D}) = repaired_U(D, FD)$.

Example 7.7. Consider the inconsistent database r and the functional dependencies FD of Example 7.6. A condensed representation of all the possible repaired relations derivable from $\langle r, FD \rangle$ is the following condensed relation $r_{\mathcal{F}D}$:

<i>emp</i>		
<i>Name</i>	<i>Dept</i>	<i>City</i>
<i>john</i>	<i>math</i>	<i>Y</i>
<i>john</i>	<i>cs</i>	<i>Y</i>
<i>bob</i>	<i>cs</i>	<i>Y</i>
<i>mary</i>	<i>physics</i>	<i>naples</i>

$$Y \in \{rome, milan, venice\}$$

as $\mathbf{G}(r_{\mathcal{F}D}) = repaired_U(r, FD)$ and the set of variables introduced in $r_{\mathcal{F}D}$ is minimal.

Theorem 7.1. *Given a database schema DS , a set FD of functional dependencies over DS and an instance D of DS , then $D_{\mathcal{F}D}$ can be computed in polynomial time. \square*

7.4 Query Answering

In this section we present a definition of probabilistic answer to queries over inconsistent databases. In particular, we first introduce the definition of probabilistic answer, where each tuple in the answer is associated with a probability

(e.g., the fraction of repaired databases from which the tuple can be derived). Next, we show how to compute probabilistic query answer by querying a probabilistic database obtained from the condensed representation of all the repaired databases. Finally, we present how to compute an approximation of such an answer in polynomial time.

Definition 7.4. *Probabilistic query answer.* Given a database D , a set FD of functional dependencies and a relational query q , the *probabilistic answer* of q over $\langle D, FD \rangle$, denoted as $q^p(D, FD)$, is defined as follows

$$q^p(D, FD) = \{ (t, p_t) \mid \exists D_i \in \text{repaired}_U(D, FD) \text{ s.t. } t \in q(D_i), \\ p_t = \frac{|\{D_i \mid D_i \in \text{repaired}_U(D, FD) \wedge t \in q(D_i)\}|}{|\text{repaired}_U(D, FD)|} \}$$

The probabilistic answer gives a set of tuples along with their probabilities, where the probability of a tuple t is defined as the percentage of the repaired databases which give t by applying q over them. It is worth noting that, unlike standard consistent answers, probabilistic answers allow us to discriminate among undefined tuples, giving them a measure of uncertainty. The tuples in a probabilistic answer can be ranked according to their probabilities, e.g. by decreasing probability.

Moreover, as the number of repaired databases can be exponential in the size of the database, the complexity of computing probabilistic answers using the formula of Definition 7.4 is also exponential. Next, we present a different method for computing probabilistic query answers over inconsistent databases.

Given two condensed tuples ct_1 and ct_2 and a substitution θ , we say that ct_1 subsumes ct_2 (or equivalently, ct_2 is an instance of ct_1) under θ , written as $ct_1 \sqsupseteq_\theta ct_2$, if $ct_2 = \theta(ct_1)$ and $\theta[1] \subseteq \text{var}(ct_1)$. Moreover, we say that $ct_1 \sqsupseteq_\theta ct_2$ if ct_2 is ground. Observe that for any two distinct tuples ct_1 and ct_2 , $ct_1 \sqsupseteq_\theta ct_2$ implies that $ct_2 \not\sqsupseteq_\theta ct_1$.

Lemma 7.1. *Let $r_{\mathcal{F}D}$ be a condensed relation derived from r and FD . For each pair of distinct condensed tuples $ct_1, ct_2 \in r_{\mathcal{F}D}$ there do not exist two substitutions θ_1 and θ_2 and a ground tuple t such that $ct_1 \sqsupseteq_{\theta_1} t$ and $ct_2 \sqsupseteq_{\theta_2} t$. \square*

The above lemma states that a ground tuple t cannot be derived from two distinct condensed tuples. The following definition introduces the concept of probabilistic relation derived from a (possibly inconsistent) relation.

Definition 7.5. *Derivation of probabilistic relations.* Let r be a relation and FD a set of functional dependencies over a relation schema $R(W)$. Let $r_{\mathcal{F}D}$ be the condensed representation for $\langle r, FD \rangle$, then $r_{\mathcal{F}D}^p$ denotes the probabilistic relation derived from $r_{\mathcal{F}D}$ as follows:

$$r_{\mathcal{F}D}^p = \{ \langle t, e_t \rangle \mid \exists ct \in r_{\mathcal{F}D} \wedge \exists \theta \text{ s.t. } ct \sqsupseteq_\theta t \wedge e_t = \bigwedge_{X/c \in \theta} X/c \}$$

where $p(e_t) = p(\bigwedge_{X/c \in \theta} X/c)$ is computed by considering the standard probabilistic function and assuming that

- two events X/c_1 and X/c_2 , where $c_1 \neq c_2$, are disjoint;
- two events X/c_1 and Y/c_2 , with $X \neq Y$, are independent;
- $Pr(X/c) = \frac{1}{|dom(X)|}$.

Observe that, as said before, for deterministic tuples (i.e. probabilistic tuples $\langle t, e_t \rangle$ such that e_t is empty), it is assumed that $e_t = \top$ so that $Pr(\top) = 1$. We recall that a condensed representation $r_{\mathcal{F}_D}$ of a set of repaired relations contains variables in place of uncertain values. In order to obtain a consistent repaired relation from $r_{\mathcal{F}_D}$, for each variable we have to replace every occurrence of it with the same value (taken from its domain). This consideration is reflected by the first assumption in the previous definition. The second assumption states that values assigned to different variables are independent. As a variable V has $n = |dom(V)|$ possible values, the probability of an event V/c (i.e. the value c is assigned to the variable V) is $\frac{1}{n}$, as stated by the last assumption in the above definition.

It is worth noting that the definition above does not take into account the number of occurrences of a value. In order to also consider the number of occurrences of values, the above probability function could be rewritten as $Pr(X/c) = \#c_X / \#X$, where $\#c_X$ is the number of occurrences of c in the source relation corresponding to X in the condensed relation and $\#X$ is the number of occurrence of X in the condensed relation. Clearly, if each value occurs once the probability function coincides with the one of Definition 7.5.

Example 7.8. Consider the database schema $r(A, B, C)$ with the functional dependency $fd = A \rightarrow B$ and the instance $R = \{r(a_1, b_1, c_1), r(a_1, b_2, c_2), r(a_1, b_1, c_3)\}$. The two repaired databases $R_1 = \{r(a_1, b_1, c_1), r(a_1, b_1, c_2), r(a_1, b_1, c_3)\}$ and $R_2 = \{r(a_1, b_2, c_1), r(a_1, b_2, c_2), r(a_1, b_2, c_3)\}$ are obtained by replacing, respectively, the unique occurrence of b_2 with b_1 and the two occurrences of b_1 with b_2 . As the derived condensed relation is $R_{fd} = \{r(a_1, X, c_1), r(a_1, X, c_2), r(a_1, X, c_3)\}$ with $X \in \{b_1, b_2\}$, we have that $Pr(X/b_1) = 2/3$ and $Pr(X/b_2) = 1/3$.

Given a probabilistic relation $r_{\mathcal{F}_D}^p, r_{\mathcal{F}_D}^b$ denotes the set of tuples $\{t | \langle t, e_t \rangle \in r_{\mathcal{F}_D}^p\}$. Given a condensed representation $D_{\mathcal{F}_D}$ of all the repaired databases for $\langle D, FD \rangle$, the probabilistic database derived from $D_{\mathcal{F}_D}$ will be denoted by $D_{\mathcal{F}_D}^p$, whereas $D_{\mathcal{F}_D}^b$ denotes the set of relations $\{r_{\mathcal{F}_D}^b | r_{\mathcal{F}_D}^p \in D_{\mathcal{F}_D}^p\}$.

Corollary 7.1. *Let $r_{\mathcal{F}_D}^p$ be a probabilistic relation derived from r and FD . For each tuple $\langle t, e_t \rangle \in r_{\mathcal{F}_D}^p$ there does not exist a tuple $\langle t', e_{t'} \rangle \in r_{\mathcal{F}_D}^p$ such that $t' = t$ and $e_{t'} \neq e_t$. \square*

The above corollary, which is straightforwardly derived from Lemma 7.1, states that in probabilistic relations (derived from inconsistent databases) there do not exist two tuples which differ only in the event part.

Example 7.9. Consider the inconsistent relation *affiliation* and the functional dependencies *fd* of Example 7.1. A condensed representation of all the repaired databases for $\langle \textit{affiliation}, \{\textit{fd}\} \rangle$ is as follows:

$$\textit{affiliation}_{fd}$$

<i>Emp</i>	<i>Dept</i>	<i>City</i>
<i>john</i>	<i>cs</i>	<i>X</i>
<i>bob</i>	<i>cs</i>	<i>X</i>

$$X \in \{\textit{rome}, \textit{milan}\}$$

The above condensed relation can be “expanded” into the following probabilistic relation:

$$\textit{affiliation}_{fd}^p$$

<i>Emp</i>	<i>Dept</i>	<i>City</i>	
<i>john</i>	<i>cs</i>	<i>rome</i>	<i>X/rome</i>
<i>john</i>	<i>cs</i>	<i>milan</i>	<i>X/milan</i>
<i>bob</i>	<i>cs</i>	<i>rome</i>	<i>X/rome</i>
<i>bob</i>	<i>cs</i>	<i>milan</i>	<i>X/milan</i>

where *X/rome* and *X/milan* are disjoint events and the probability of each of them is 0.5. The relation $\textit{affiliation}_{fd}^b$ is equal to the projection of $\textit{affiliation}_{fd}^p$ over the attributes *Emp*, *Dept* and *City*.

Theorem 7.2. *Given a database D and a set of functional dependencies FD , $\textit{repaired}_U(D, FD) = \textit{repaired}_{id}(D_{\mathcal{FD}}^b, FD)$. \square*

The previous theorem states that the repaired databases for $\langle D, FD \rangle$ obtained by means of tuple updates, can be computed by repairing the database $D_{\mathcal{FD}}^b$ by means of insertion and deletion of tuples.

Theorem 7.3. *Given a database D , a set FD of functional dependencies and a relational query q , then*

$$q^p(D, FD) = q(D_{\mathcal{FD}}^p) \quad \square$$

The previous theorem states that given a database D , a set FD of functional dependencies and a relational query q , the probabilistic query answer $q^p(D, FD)$ can be computed as follows:

- firstly, a condensed representation of all the repaired databases for $\langle D, FD \rangle$, namely $D_{\mathcal{FD}}$, is derived;
- next, $D_{\mathcal{FD}}$ is converted into a probabilistic database $D_{\mathcal{FD}}^p$;
- finally, the intensional evaluation of q over $D_{\mathcal{FD}}^p$ is computed and probabilities to each tuple in the answer are assigned.

As computing the probability of an event e of an answer tuple is a #P-complete problem, next we present an approach for computing approximate probabilistic answers in polynomial time.

Approximate Probabilistic Query Answering.

In this section we consider positive relational queries, that is queries using the relational operators σ , π , \times and \cup . Moreover, we assume that PRA operators compute events written as DNF formulae, that is events are of the form $C_1 \vee \dots \vee C_n$, where each C_i is a conjunction of events of the form X/c_X and events appearing more than one time in a conjunction are considered once.

Given an event $e = C_1 \vee \dots \vee C_n$, its probability $Pr(e)$ can be computed by applying the well-known inclusion-exclusion formula, i.e.

$$Pr(e) = \sum_{k=1}^n (-1)^{k+1} \sum_{1 \leq i_1 < \dots < i_k \leq n} Pr(C_{i_1} \wedge \dots \wedge C_{i_k})$$

where the probability of a conjunction of events $C = e_1 \wedge \dots \wedge e_k$ is equal to $Pr(e_1) \times \dots \times Pr(e_k)$ if C does not contain two events X/c_1 and X/c_2 with $c_1 \neq c_2$, otherwise it is equal to 0.

The issue of approximating an inclusion-exclusion formula has been dealt with in [61, 57]. In particular, a method to approximate an inclusion-exclusion formula in polynomial time has been proposed in [57]. Specifically, given an event $e = C_1 \vee \dots \vee C_n$ and the probabilities of all the j -wise conjunctions of C_i for $j = 1..k$, $Pr(e)$ can be approximated with an error of $e^{-\Omega(\frac{k^2}{n \log n})}$. We denote by $Pr_k(e)$ the so obtained approximation of $Pr(e)$.

Definition 7.6. *Approximate probabilistic answer.* Given a database D , a set FD of functional dependencies and a relational query q , the k -approximate probabilistic answer of q over $\langle D, FD \rangle$, denoted as $AQ_k(D, FD)$, is defined as follows

$$AQ_k(D, FD) = \{ \langle t, a_t \rangle \mid \exists \langle t, e_t \rangle \in q(D_{\mathcal{F}D}^p) \text{ and } a_t = Pr_k(e_t) \} \quad \square$$

Theorem 7.4. *Given a database D , a set FD of functional dependencies and a relational query q , the k -approximate probabilistic answer of q over $\langle D, FD \rangle$ can be computed in polynomial time.* \square

7.5 Discussion

In this chapter we have presented a framework for querying inconsistent databases where constraints consist of (particular sets of) functional dependencies. Inconsistent databases are repaired by means of tuple updates, rather than tuple deletions, in order to preserve better their information. Moreover, query answers are “probabilistic”, that is they are tuples associated with probabilities: the probability of an answer depends on the number of repaired databases that allow us to obtain the answer. This semantics of query answering allows us to discriminate among answers which are not consistent thus providing

more informative query answers. We have proposed an algorithm to compute probabilistic query answers. This algorithm allows us to compute approximate probabilistic query answers in polynomial time.

Andritsos et al. [2] presented an approach for querying dirty databases containing duplicate tuples (i.e. inconsistent databases that violates a set of key constraints) where each duplicate is associated with a probability of being in the clean database. A technique for querying dirty databases is proposed. It consists in rewriting a query into an SQL query that computes each answer with the probability that the answer is in the clean database. The rewriting cannot be obtained in general as it is applicable only to a special class of select-project-join queries, called *rewritable queries*. The main difference between the approach presented in this chapter and the one introduced in [2] is that we consider a more general framework, namely a special class of functional dependencies and positive relational algebra queries, and compute approximate probabilistic answers (i.e. answers whose associated probabilities are approximated), whereas the technique proposed by Andritsos et al. computes (exact) probabilistic answers for more restricted constraints and queries (key constraints and a subset of SPJ queries).

In [30] it has been shown that for every conjunctive query, the complexity of evaluating it on a probabilistic database is either PTIME or #P-complete, and an algorithm for deciding whether a given conjunctive query is PTIME or #P-complete is given. The problem of querying and managing probabilistic databases has been dealt with also in [29, 31].

An approach for repairing inconsistent databases by means of tuple updates has been proposed in [76, 14, 77]. Specifically, [76] presents a notion of update-based repairing, and the construction of single databases, called *nuclei*, that can replace all (possibly infinitely many) repaired databases for the purpose of consistent query answering. The construction of nuclei for full dependencies and conjunctive queries is shown. Consistent query answering and constructing nuclei is generally intractable under update-based repairing. In [14] an approach for repairing inconsistent databases is proposed. In such a framework, a database which violates a set of functional and inclusion dependencies is repaired by modifying attribute values and by inserting new tuples. Each update operation has a cost. As finding a repaired database with minimum cost in this model is NP-complete, a heuristic approach is proposed.

Conclusions

Although integrity constraints have long been used to maintain database consistency, nowadays there are plenty of scenarios where inconsistency arises because integrity constraints may not be enforced or satisfied. The problem of extracting reliable information from inconsistent databases has been extensively studied in the past years. Most of the works in the literature are based on the *consistent query answering* (CQA) framework. This framework relies on the notions of *repair* and *consistent query answer*. A repair for an inconsistent database is a consistent database which is as close as possible to the original one. The *consistent answers* to a query over a possibly inconsistent database are those answers that can be obtained from every repair.

In this thesis, we have addressed several issues regarding the problem of repairing and querying inconsistent databases.

First, we have dealt with the problem of expressing preferences among repairs. The motivation of this work stems from the observation that as an inconsistent database can be repaired in different ways, it is natural to express preferences among the possible actions which restore consistency. In this regard, we have proposed a logical framework based on *prioritized active integrity constraints* (PAICs). A PAIC allows us to express a universal integrity constraints, the feasible updates which should be performed whenever the constraint is violated and preferences among the feasible updates. These preferences determine preferences also among repairs, so that *preferred* repairs can be selected among all the possible repairs. The preferred repairs are the only ones to be considered during query answering. We have studied some desirable properties on the set of preferred repairs which hold in the proposed framework. It has been shown that prioritized active integrity constraints can be rewritten into disjunctive Datalog programs so that repairs correspond to stable models.

Inconsistency leads to *uncertainty* as to the actual values of tuple attributes. Thus, it is natural to study the possible use of incomplete database frameworks in this context. The set of repairs for a possibly inconsistent database could be represented by means of an incomplete database whose

possible worlds are exactly the repairs of the inconsistent database. We have studied this issue by considering a specific incomplete database framework: *disjunctive databases*. Thus we have addressed the problem of *representing* the set of repairs of a possibly inconsistent database by means of a disjunctive database, i.e. a disjunctive database whose minimal models are the repairs. We have shown that, given a database and a set of denial constraints, there exists a (unique) disjunctive database, called *canonical*, which represents the repairs of the database w.r.t. the constraints and is contained in any other disjunctive database with the same set of minimal models. Moreover, we have proposed an algorithm for computing the canonical disjunctive database. We have also studied the size of the canonical disjunctive database in the presence of functional dependencies for both set- and card-repairs.

We have proposed a framework for repairing and querying relational databases which may be inconsistent with respect to functional dependencies and foreign key constraints. In order to restore the consistency of inconsistent databases, we have proposed a repairing strategy that performs tuple insertions when foreign key constraints are violated and tuple updates when functional dependency violations occur (tuple deletions are never performed). Since tuple insertions and updates may introduce, respectively, null and unknown values in the database, we have proposed a semantics of constraint satisfaction for databases containing null and unknown values. Our approach always allows us to obtain a unique (up to renaming of unknown and null values) repaired database which can be computed in polynomial time. The result of the repairing technique is an incomplete database (in particular, an OR-database). The semantics of query answering over an inconsistent database consists in computing the certain query answers on the repaired database. We have identified a class of conjunctive queries whose answers can be computed in polynomial time.

Computing consistent query answers is in general an intractable problem. Different restricted classes of queries and constraints for which the problem is tractable have been identified. In this thesis we tackled this problem by providing techniques which allows us to compute “approximate” query answers over inconsistent databases in polynomial time in the presence of general classes of queries and constraints.

Specifically, we have proposed a technique for computing a sound and incomplete set of consistent query answers in polynomial time. The proposed approach relies on a *three-valued repairing strategy* where update operations make the truth value of database atoms *true*, *false* or *undefined*. Thus, in this setting, three-valued databases are considered and a new semantics of constraint satisfaction (for three-valued databases) has been proposed. We have shown that the set of three-valued repairs defines a lower semi-lattice whose top elements are standard repairs (performing tuple deletions and insertions only) and whose bottom element is called *deterministic repair*. We have shown that by evaluating a query over the deterministic repair we get sound, but not complete, consistent answers. Moreover, we have studied some

classes of queries and constraints for which the proposed technique is also complete. It has been shown also that the deterministic repair and query answers can be computed in polynomial time, by means of a stratified Datalog program derived from the integrity constraints.

Finally, we have proposed a framework for querying inconsistent databases which aims at preserving better the information in an inconsistent database and providing more informative query answers. In order to achieve these goals, we have adopted notions of repair and query answer which differ from the classical ones. Specifically, the repairing strategy relies on value-updates whereas answers to queries are tuples associated with probabilities. We have proposed a technique for computing approximate probabilistic query answers in polynomial time.

References

1. Abiteboul, S., Hull, R., and Vianu, V., *Foundations of Databases*. Addison-Wesley, 1995.
2. Andritsos, P., Fuxman, A., and Miller, R. J., Clean Answers over Dirty Databases: A Probabilistic Approach. *International Conference on Data Engineering (ICDE)*, 2006.
3. Antova, L., Koch, C., and Olteanu, D., 10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information. *International Conference on Data Engineering (ICDE)*, pp. 606–615, 2007.
4. Arenas, M., Bertossi, L. E., and Chomicki, J., Consistent Query Answers in Inconsistent Databases. *ACM Symposium on Principles of Database Systems (PODS)*, pp. 68–79, 1999.
5. Arenas, M., Bertossi, L. E., and Chomicki, J., Scalar Aggregation in FD-Inconsistent Databases. *International Conference on Database Theory (ICDT)*, pp. 39–53, 2001.
6. Arenas, M., Bertossi, L. E., and Chomicki, J., Specifying and Querying Database Repairs using Logic Programs with Exceptions. *International Conference on Flexible Query Answering Systems (FQAS)*, pp. 27–41, 2000.
7. Arenas, M., Bertossi, L. E., and Chomicki, J., Answer Sets for Consistent Query Answering in Inconsistent Databases. *Theory and Practice of Logic Programming*, Vol. 3(4–5), pp. 393–424, 2003.
8. Arenas, M., Bertossi, L. E., Chomicki, J., He, X., Raghavan, V., and Spinrad, J., Scalar aggregation in inconsistent databases. *Theoretical Computer Science*, Vol. 3(296), pp. 405–434, 2003.
9. Baral, C., Kraus, S., Minker, J., and Subrahmanian, V. S., Combining Knowledge Bases Consisting of First-Order Theories. *Computational Intelligence*, Vol. 8, pp. 45–71, 1992.
10. Benjelloun, O., Sarma, A. D., Halevy, A. Y., Theobald, M., and Widom, J., Databases with uncertainty and lineage. *VLDB Journal*, Vol. 17(2), pp. 243–264, 2008.
11. Bertossi, L., Consistent Query Answering in Databases. *SIGMOD Record*, Vol. 35(2), pp. 68–76, 2006.
12. Bertossi, L., Bravo, L., Franconi, E., and Lopatenko, A., Complexity and Approximation of Fixing Numerical Attributes in Databases Under Integrity

- Constraints. *International Symposium on Database Programming Languages (DBPL)*, pp. 262–278, 2005.
13. Bertossi, L., and Chomicki, J., Query answering in inconsistent databases. *Logics for Emerging Applications of Databases*, pp. 43–83, 2003.
 14. Bohannon, P., Flaster, M., Fan, W., and Rastogi, R., A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. *ACM SIGMOD International Conference on Management of Data*, pp. 143–154, 2005.
 15. Borgida, A., Language Features for Flexible Handling of Exceptions in Information Systems. *ACM Transactions on Database Systems*, Vol. 10(4), pp. 565–603, 1985.
 16. Bravo, L., and Bertossi, L. E., Semantically Correct Query Answers in the Presence of Null Values. *EDBT Workshop on Inconsistency and Incompleteness in Databases (IIDB)*, pp. 336–357, 2006.
 17. Brewka, G., Niemela, I., and Truszczynski, M., Answer Set Optimization. *International Joint Conference on Artificial Intelligence (IJCAI)* pp. 867–872, 2003.
 18. Bry, F., Query Answering in Information Systems with Integrity Constraints. *Working Conference on Integrity and Control in Information Systems (IICIS)*, pp. 113–130, 1997.
 19. Cali, A., Lembo, D., and Rosati, R., On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. *ACM Symposium on Principles of Database Systems (PODS)*, pp. 260–271, 2003.
 20. Cali, A., Lembo, D., and Rosati, R., Query rewriting and answering under constraints in data integration systems. *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 16–21, 2003.
 21. Caroprese, L., Greco, S., and Molinaro, C., Prioritized Active Integrity Constraints for Database Maintenance. *Database Systems for Advanced Applications (DASFAA)*, pp. 459–471, 2007.
 22. Caroprese, L., Greco, S., Sirangelo, C., and Zumpano, E., Declarative Semantics of Production Rules for Integrity Maintenance. *International Conference on Logic Programming (ICLP)* pp. 26–40, 2006.
 23. Caroprese, L., Greco, S., and Zumpano, E., Active Integrity Constraints for Database Consistency Maintenance. *IEEE Transactions on Knowledge and Data Engineering*, accepted for publication, 2008.
 24. Chomicki, J., Consistent Query Answering: Five Easy Pieces. *International Conference on Database Theory (ICDT)*, pp. 1–17, 2007.
 25. Chomicki, J., and Marcinkowski, J., Minimal-Change Integrity Maintenance Using Tuple Deletions. *Information and Computation*, Vol. 197(1–2), pp. 90–121, 2005.
 26. Chomicki, J., Marcinkowski, J., and Staworko, S., Computing Consistent Query Answers Using Conflict Hypergraphs. *International Conference on Information and Knowledge Management (CIKM)*, pp. 417–426, 2004.
 27. Chomicki, J., Marcinkowski, J., and Staworko, S., Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. *International Conference on Extending Database Technology (EDBT)*, System demo, pp. 841–844, 2004.
 28. Chomicki, J., Staworko, S., and Marcinkowski, J., Preference-Driven Querying of Inconsistent Relational Databases. *International Workshop on Inconsistency and Incompleteness in Databases*, pp. 318–335, 2006.
 29. Dalvi, N., and Suciu, D., Management of probabilistic data: foundations and challenges. *ACM Symposium on Principles of Database Systems (PODS)*, pp. 1–12, 2007.

30. Dalvi, N., and Suciu, D., The Dichotomy of Conjunctive Queries on probabilistic Structures. *ACM Symposium on Principles of Database Systems (PODS)*, pp. 293–302, 2007.
31. Dalvi, N., and Suciu, D., Efficient Query Evaluation on Probabilistic Databases. *International Conference on Very Large Data Bases (VLDB)*, pp. 864–875, 2004.
32. Delgrande, J. P., Schaub, T., and Tompits, H., A Framework for Compiling Preferences in Logic Programs. *Theory and Practice of Logic Programming*, Vol. 3(2), pp. 129–187, 2003.
33. Dey, D., and Sarkar, S., A Probabilistic Relational Model and Algebra. *ACM Transactions on Database Systems*, Vol. 21(3), pp. 339–369, 1996.
34. Fernández, J. A., and Minker, J., Semantics of disjunctive deductive databases. *International Conference on Database Theory (ICDT)*, pp. 21–50, 1992.
35. Flesca, S., Furfaro, F., and Parisi, F., Consistent Query Answers on Numerical Databases under Aggregate Constraints. *International Workshop on Database Programming Languages (DBPL)*, pp. 279–294, 2005.
36. Fuhr, N., A Probabilistic Relational Model for the Integration of IR and Databases. *International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 309–317, 1993.
37. Fuhr, N., and Rolleke, T., A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. *ACM Transactions on Information Systems*, Vol. 15(1), pp. 32–66, 1997.
38. Furfaro, F., Greco, S., and Molinaro, C., A three-valued semantics for querying and repairing inconsistent databases. *Annals of Mathematics and Artificial Intelligence*, Vol. 51(2–4), pp. 167–193, 2007.
39. Fuxman, A., Fazli, E., and Miller, R. J., ConQuer: Efficient Management of Inconsistent Databases. *ACM SIGMOD International Conference on Management of Data*, pp. 155–166, 2005.
40. Fuxman, A., Fuxman, D., and Miller, R. J., ConQuer: A System for Efficient Querying Over Inconsistent Databases. *International Conference on Very Large Data Bases (VLDB)*, System demo, pp. 1354–1357, 2005.
41. Fuxman, A., and Miller, R. J., First-Order Query Rewriting for Inconsistent Databases. *International Conference on Database Theory (ICDT)*, pp. 337–351, 2005.
42. Fuxman, A., and Miller, R. J., First-Order Query Rewriting for Inconsistent Databases. *Journal of Computer and System Sciences*, Vol. 73(4), pp. 610–635, 2007.
43. Gelfond, M., and Lifschitz, V., Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, Vol. 9(3–4), pp. 365–386, 1991.
44. Gelfond, M., and Son, T. C., Reasoning with prioritized defaults. *Logic Programming and Knowledge Representation (LPKR)*, pp. 164–223, 1997.
45. Grant, J., and Hunter, A., Measuring inconsistency in knowledgebases. *Journal of Intelligent Information Systems*, Vol. 27(2), pp. 159–184, 2006.
46. Greco, S., Sirangelo, C., Trubitsyna, I., and Zumpano, E., Preferred Repairs for Inconsistent Databases. *International Conference on Database and Expert Systems Applications (DEXA)*, pp. 44–55, 2004.
47. Greco, S., and Molinaro, C., Querying and Repairing Inconsistent Databases Under Three-Valued Semantics. *International Conference on Logic Programming (ICLP)*, pp. 149–164, 2007.

48. Greco, S., and Molinaro, C., Approximate Probabilistic Query Answering over Inconsistent Databases. *International Conference on Conceptual Modeling (ER)*, pp. 311–325, 2008.
49. Greco, S., and Zumpano, E., Querying Inconsistent Databases. *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, pp. 308–325, 2000.
50. Greco, G., Greco, S., and Zumpano, E., A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15(6), pp. 1389–1408, 2003.
51. Hunter, A., Measuring Inconsistency in Knowledge via Quasi-Classical Models. *National Conference on Artificial Intelligence (AAAI)*, pp. 68–73, 2002.
52. Hunter, A., Evaluating significance of inconsistencies. *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 468–473, 2003.
53. Hunter, A., and Konieczny, S., Approaches to Measuring Inconsistent Information. *Inconsistency Tolerance*, pp. 191–236, 2005.
54. Imielinski, T., Lipski, W. Jr., Incomplete Information in Relational Databases. *Journal of the ACM*, Vol. 31(4), pp. 761–791, 1984.
55. Imielinski, T., Naqvi, S. A., and Vadaparty, K. V., Incomplete objects - a data model for design and planning applications. *ACM SIGMOD Conference*, pp. 288–297, 1991.
56. Imielinski, T., van der Meyden, R., and Vadaparty, K. V., Complexity tailored design: A new design methodology for databases with incomplete information. *Journal of Computer and System Sciences*, Vol. 51(3), pp. 405–432, 1995.
57. Kahn, J., Linial, N., and Samorodnitsky, A., Inclusion-Exclusion: Exact and Approximate. *Combinatorica*, Vol. 16(4), pp. 465–477, 1996.
58. Kowalski, R. A., and Sadri, F., Logic Programs with Exceptions. *New Generation Computing*, Vol. 9(3–4), pp. 387–400, 1991.
59. Leone, N., Pfeifer, G., Faber, W., Calimeri, F., Dell’Armi, T., Eiter, T., Gottlob, G., Ianni, G., Ielpa, G., Koch, K., Perri, S., and Polleres, A., The DLV System. *International Conference on Logics in Artificial Intelligence (JELIA)*, pp. 537–540, 2002.
60. Levene, M., and Loizou, G., Null Inclusion Dependencies in Relational Databases. *Information and Computation* Vol. 136(2), pp. 67–108, 1997.
61. Linial, N., and Nisan, N., Approximate Inclusion-Exclusion. *ACM Symposium on the Theory of Computing (STOC)*, pp. 260–270, 1990.
62. Lin, J., and Mendelzon, A. O., Merging Databases under Constraints. *International Journal of Cooperative Information Systems*, Vol. 7(1), pp. 55–76, 1996.
63. Lopatenko, A., and Bertossi, L., Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. *International Conference on Database Theory (ICDT)*, pp. 179–193, 2007.
64. Lozinskii, E. L., Resolving Contradictions: A Plausible Semantics for Inconsistent Systems. *Journal of Automated Reasoning*, Vol. 12(1), pp. 1–32, 1994.
65. Martinez, M. V., Pugliese, A., Simari, G. I., Subrahmanian, V. S., and Prade, H., How Dirty Is Your Relational Database? An Axiomatic Approach. *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)*, pp. 103–114, 2007.
66. Minker, J., and Seipel, D., Disjunctive logic programming: A survey and assessment. *Computational Logic: Logic Programming and Beyond*, pp. 472–511, 2002.

67. Molinaro, C., Polynomial Time Queries over Inconsistent Databases. *International Conference on Scalable Uncertainty Management (SUM)*, pp. 312–325, 2008.
68. Molinaro, C., Chomicki, J., and Marcinkowski, J., Disjunctive Databases for Representing Repairs. Submitted to a journal, 2008.
69. Rao, P., Sagonas, K. F., Swift, T., Warren, D. S., and Freire, J., XSB: A System for Efficiently Computing WFS. *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pp. 431–441, 1977.
70. Sakama, C., and Inoue, K., Prioritized Logic Programming and Its Application to Commonsense Reasoning. *Artificial Intelligence*, Vol. 123(1–2), pp. 185–222, 2000.
71. Staworko, S., and Chomicki, J., Consistent Query Answering in the Presence of Universal Constraints. Technical report [arxiv: 0809.1551v1](https://arxiv.org/abs/0809.1551v1) [cs.DB], 2008.
72. Syrjänen, T., and Niemelä, I., The Smodels System. *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pp. 434–438, 2001.
73. Ullman, J., *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Vol. I, 1988.
74. Van Gelder, A., Ross, K. A., and Schlipf, J. S., The Well-Founded Semantics for General Logic Programs. *Journal of ACM*, Vol. 38(3), pp. 620–650, 1991.
75. Vardi, M. Y., The Complexity of Relational Query Languages. *ACM Symposium on Theory of Computing (STOC)*, pp. 137–146, 1982.
76. Wijsen, J., Database repairing using updates. *ACM Transactions on Database Systems*, Vol. 30(3), pp. 722–768, 2005.
77. Wijsen, J., Project-Join-Repair: An Approach to Consistent Query Answering Under Functional Dependencies. *International Conference on Flexible Query Answering Systems (FQAS)*, pp. 1–12, 2006.
78. Zaniolo, C., Database Relations with Null Values. *Journal of Computer and System Sciences*, Vol. 28(1), pp. 142–166, 1984.
79. Zaniolo, C., Database Relations with Null Values. *ACM Symposium on Principles of Database Systems (PODS)*, pp. 27–33, 1982.
80. Zhang, Y., and Foo, N., Answer sets for prioritized logic programs. *Symposium on Logic Programming*, pp. 69–83, 1997.