

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
XIX ciclo

Tesi di Dottorato

Querying Inconsistent Data: Repairs and Consistent Answers

Francesco Parisi



UNIVERSITÀ DELLA CALABRIA

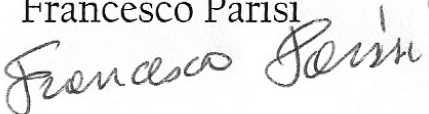
Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica

XIX ciclo

Tesi di Dottorato

Querying Inconsistent Data:
Repairs and Consistent Answers

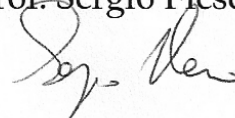
Francesco Parisi



Coordinatore
Prof. Domenico Talia



Supervisore
Prof. Sergio Flesca



DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
Settore Scientifico Disciplinare: ING-INF/05

to Ale

Preface

A shared opinion in the computer science community is that inconsistency is undesirable. It is a common belief that databases and knowledge bases should be completely free of inconsistency. Integrity constraints are the mechanism employed in databases to guarantee that available data correctly model the outside world. An integrity constraint can be considered as a boolean query which must always be true. The traditional approach, implemented by commercial database management systems, is to avoid inconsistency by aborting updates or transactions yielding to an integrity constraint violation.

In some contexts integrity constraints satisfaction cannot be guaranteed employing the traditional approach. For instance, when knowledge from multiple sources is integrated, as in the contexts of data warehousing, database integration and automated reasoning systems, it is not possible to guarantee the consistency on the integrated information applying the traditional approach. More in detail, when different source databases are integrated together, although every source database is consistent with respect to a given set of integrity constraints, in the resulting integrated database many different kinds of discrepancies may arise. In particular, possible discrepancies are due to (i) different sets of integrity constraints that are satisfied by different sources, and (ii) constraints that may be globally violated, even if every source database locally satisfies the same integrity constraints. In this case there is no update that can be rejected in order to guarantee the consistency of the integrated database as this database instance is not resulting from an update performed on a source database, but from merging multiple independent data sources. For instance, if we know that a person should have a single address but multiple data sources contain different addresses for the same person, it is not clear how to resolve this violation through aborting some update.

The approach for dealing with inconsistency in contexts such as database integration is that of “accepting” it and providing appropriate mechanisms to handle inconsistent data. In literature the process achieving a consistent state in a database with respect to a given set of constraints is considered as a separate process that can be executed after that inconsistency is detected.

In other words, as opposed to traditional approaches, the key idea to inconsistency handling is to live with an inconsistent database, modifying query semantics in order to obtain only consistent information as answers to queries posed on inconsistent databases. Indeed, when a query is posed on an inconsistent database (according to the traditional semantics) it is possible to yield information which are not consistent with respect to the integrity constraints. Consistent information is the one that is invariant or persists under all possible “minimal” ways of restoring consistency of the database. Indeed, the restoration of consistency should be accomplished with a minimal impact on the original inconsistent database, trying to preserve as much information as possible. Every database instance corresponding to a “minimal” way of restoring the consistency is called *repair*. There may be several alternative minimal repairs for a database. Thus, what results true w.r.t. an inconsistent database instance is what results true w.r.t. all “repaired” instances.

The following example describes a typical inconsistency-prone data integration scenario: data from two consistent source databases are integrated, and the resulting database turns out to be in an inconsistent state.

Example 1 Consider the following database scheme consisting of the single binary relation $Teaches(Course, Professor)$, where the attribute $Course$ is a key for $Teaches$. Assume there are two different instances of $Teaches$ as reported in the following figure.

<i>Course</i>	<i>Professor</i>
<i>CS</i>	<i>Mary</i>
<i>Math</i>	<i>John</i>

<i>Course</i>	<i>Professor</i>
<i>CS</i>	<i>Frank</i>
<i>Math</i>	<i>John</i>

Each of the two instances satisfy the key constraint but, from their union we derive the inconsistent relation shown in the following figure. This relation does not satisfy the constraint since there are two distinct tuples with the same value for the attribute $Course$.

<i>Course</i>	<i>Professor</i>
<i>CS</i>	<i>Mary</i>
<i>Math</i>	<i>John</i>
<i>CS</i>	<i>Frank</i>

We can consider as minimal ways for restoring consistency of the database, i.e. repairs for this database, the following two relations obtained by deleting one of the two tuples that violate the key constraint.

<i>Course</i>	<i>Professor</i>
<i>Math</i>	<i>John</i>
<i>CS</i>	<i>Frank</i>

<i>Course</i>	<i>Professor</i>
<i>CS</i>	<i>Mary</i>
<i>Math</i>	<i>John</i>

On the basis of these alternative repairs, what is consistently true is that ‘*John*’ is the teacher the course ‘*Math*’ and that there is a course named ‘*CS*’.

□

The approach for handling inconsistency described in the example above shares many similarities with the problem of updating a database seen as a logical theory by means of a set of sentences (the integrity constraints). Specifically, given a knowledge base K and a revision α , belief revision theory is concerned with the properties that should hold for a rational notion of updating K with α . If $K \cup \alpha$ is inconsistent, then belief revision theory assumes the requirement that the knowledge should be revised so that the result is consistent. In the database case, the data are flexible, subject to repair, but the integrity constraints are hard, not to be given up.

The “revision” of a database instance by the integrity constraints produces new database instances, i.e. repairs for the original database. Therefore, what is consistently true is what is true with respect to every repaired database. The notion of a fact which is consistently true corresponds to the notion of inference, called *counterfactual inference*, used in the belief revision community. In the database community, the concept of fact which is consistently true has been formalized with the notion of *consistent query answer*, i.e. an answer to a query which results true in every repaired database. Consistent query answer provides a conservative “lower bound” on the information contained in a database.

Example 1 (continued) Consider the query $Q(x, y) = Teaches(x, y)$ which intends to retrieve the names of courses with their relative teachers. Obviously, if Q is directly posed on the inconsistent database it returns answers which are consistent with the key constraints and others which are not.

On the other hand, the consistent query answers for Q are those which would be returned posing the query in every repaired database, i.e. the tuple $\langle Math, John \rangle$.

□

A repair for an inconsistent database (with respect to a set of integrity constraints) is a consistent database which is “as much close as possible” to the original instance. Different notions of closeness can be defined, each of them corresponding to a different repair notion. For instance, in the example above repairs are obtained by performing minimal sets of insertion and deletion of (whole) tuples on the original database, so that the resulting database

satisfies the integrity constraints. Another possible notion of repair is that allowing updates of values within some tuples. Considering the example above, this means that the value of the attribute *Professor* in one of the two conflicting tuples must be changed in such a way that a consistent status is obtained. Specifically, we may update either the value ‘*Mary*’ to ‘*Frank*’ in the tuple $\langle CS, Mary \rangle$ or the value ‘*Frank*’ to ‘*Mary*’ in the tuple $\langle CS, Frank \rangle$, obtaining again the two repairs shown in Example 1.

In general, different set of repairs can be obtained under different repair notions. Further, since the set of consistent answers to a query posed on an inconsistent database depends on the set of repairs for the database, the repair semantics also alters the set of consistent query answers.

As shown in the following example, when tuples contain both correct and erroneous components the two repair semantics discussed above do not coincide.

Example 2 Consider the following database scheme consisting of the relation $Employee(Code, Name, Salary)$, where the attribute *Code* is a key for the relation. Assume that the constraint $\forall x, y, z \neg[Employee(x, y, z) \wedge z < 10000]$ is defined, stating that each employee must have salary greater than 10000.

Consider the following (inconsistent) instance for the relation *Employee*.

<i>Code</i>	<i>Name</i>	<i>Salary</i>
111	<i>John</i>	1000

Under the repair semantics of deletion/insertion of tuples, there is a unique repair for the relation *Employee*: the empty database instance. On the other hand, if the repaired database is obtained by changing attribute values, there are infinitely many repairs, each of them containing a tuple of the form $\langle 111, John, c \rangle$, where c a constant greater than or equal to 10000.

Thus, under the latter repair notion the consistent answer to the query asking for the existence of the employee with code ‘111’ is *yes*, whereas under the former repair notion the consistent answer is *no*. This happens because when we delete a tuple because it contains an error, we also lose the correct components as an undesirable side effect.

□

Several theoretical issues regarding the consistent query answers problem have been widely investigated in literature and some techniques for evaluating consistent answers have been proposed too. The problem of computing consistent answers has been studied among several dimensions, such as the repair semantics, the classes of queries and constraints. Many approaches in literature assume that tuple insertions and deletions are the basic primitives

for repairing inconsistent data. More recently, repairs consisting also of value-update operations have been considered. The complexity of computing consistent answers for different classes of first-order queries and aggregate queries has been investigated in presence of several classes of integrity constraints. All previous works in this area deal with “classical” forms of constraint (such as keys, foreign keys, functional dependencies), and propose different strategies for updating inconsistent data reasonably, in order to make it consistent by means of minimal changes. Indeed these kinds of constraint often do not suffice to manage data consistency, as they cannot be used to define algebraic relations between stored values. In fact, this issue frequently occurs in several scenarios, such as scientific databases, statistical databases, and data warehouses, where numerical values of tuples are derivable by aggregating values stored in other tuples.

In this thesis, we first provide a comprehensive survey of the techniques for repairing and querying inconsistent databases. Then, we focus on databases storing data that may violate a set of *aggregate constraints*, i.e. integrity constraints defined on aggregate values extracted from the database. These constraints are defined on numerical attributes (such as sale prices, costs, etc.) which represent measure values and are not intrinsically involved in other forms of constraints.

Example 3 Table 1 represents a *cash budget* for a firm, that is a summary of cash flows (receipts, disbursements, and cash balances) over a specific period (typically, a year). Values ‘*det*’, ‘*aggr*’ and ‘*drv*’ in column *Type* stand for *detail*, *aggregate* and *derived*, respectively. In particular, an item of the table is *aggregate* if it is obtained by aggregating items of type *detail* of the same section, whereas a *derived* item is an item whose value can be computed using the values of aggregate items belonging to any section.

<i>Section</i>	<i>Subsection</i>	<i>Type</i>	<i>Value</i>
Receipts	cash sales	det	100
Receipts	receivables	det	120
Receipts	total cash receipts	aggr	250
Disbursements	payment of accounts	det	120
Disbursements	capital expenditure	det	0
Disbursements	long-term financing	det	40
Disbursements	total disbursements	aggr	160
Balance	net cash inflow	drv	60

Table 1. A cash budget

A cash budget must satisfy the following integrity constraints:

1. for each section, the sum of the values of all *detail* items must be equal to the value of the *aggregate* item of the same section;

2. the *net cash inflow* must be equal to the difference between *total cash receipts* and *total disbursements*.

Table 1 was acquired by means of an Optical Character Recognition (OCR) tool from a paper document, reporting the cash budget for a specific period. The original paper document was consistent, but some symbol recognition errors occurred during the digitizing phase, as constraints 1) and 2) are not satisfied on the acquired data, that is:

- i) in section *Receipts*, the aggregate value of *total cash receipts* is not equal to the sum of detail values of the same section.
- ii) the value of *net cash inflow* is not to equal the difference between *total cash receipts* and *total disbursements*.

In order to exploit the digital version of the cash budget, a fundamental issue is to define a reasonable strategy for locating OCR errors, and then repairing the acquired data to extract reliable information. □

Most of well-known techniques for repairing data violating constraints accomplish this task by performing deletions and insertions of tuples, as shown in Example 1. More recently, value-update operations have been exploited in some techniques for restoring consistency, as shown in Example 2. Indeed, the former repairing strategy is not suitable for contexts analogous to that of Example 3, that is of data acquired by OCR tools from paper documents. For instance, repairing Table 1 by either adding or removing rows means hypothesizing that the OCR tool either jumped a row or “invented” it when acquiring the source paper document, which is rather unrealistic. The same issue arises in other scenarios dealing with numerical data representing pieces of information acquired automatically, such as sensor networks. In a sensor network with error-free communication channels, no reading generated by sensors can be lost, thus repairing the database by adding new readings (as well as removing collected ones) is of no sense. In this kind of scenario, the most natural approach to data repairing is updating directly the numerical data: this means working at attribute-level, rather than at tuple-level. For instance, in the case of Example 3, we can reasonably assume that inconsistencies of digitized data are due to symbol recognition errors, and thus trying to re-construct actual data values is well founded.

Main Contributions

The main contributions of the thesis are the following.

- 1) Recently, some works provide an overview of some issues related to the computation of consistent query answers in inconsistent databases [15, 31, 30]. In this dissertation we provide an extensive survey of the techniques for

repairing and querying inconsistent relational databases. We distinguish four parameters for classifying and comparing of the existing techniques. First, we discern two *repairing paradigms*, namely the *tuple-based* and the *attribute-based* repairing paradigm. According to the former paradigm a repair for a database is obtained by inserting and/or deleting tuples, whereas according to the latter a repair is obtained by (also) modifying attribute values within tuples. Second, we distinguish several *repair semantics* which entail different orders among the set of consistent database instances that can be obtained for an inconsistent database with respect to a given set of integrity constraints. Third, we classify the techniques on the basis of the *classes of queries* considered for computing consistent answers. Finally, we compare the different approaches in literature on basis of the *classes of integrity constraints* which are assumed to be defined on the database.

- 2) We investigate the problem of repairing and extracting reliable information from data violating a given set of *aggregate constraints*. These constraints consist of linear inequalities on aggregate-sum queries issued on measure values stored in the database. This syntactic form enables meaningful constraints to be expressed. Indeed, aggregate constraints frequently occur in many real-life scenarios where guaranteeing the consistency of numerical data is mandatory.

We consider database repairs consisting of sets of value-update operations aiming at re-constructing the correct measure values of inconsistent data. We adopt two different criteria for determining whether a set of update operations repairing data can be considered “reasonable” or not: *set-minimal semantics* and *card-minimal semantics*. Both these semantics aim at preserving the information represented in the source data as much as possible. They correspond to different repairing strategies which turn out to be well-suited for different application scenarios.

We provide the complexity characterization of three fundamental problems: (i) *repairability*: is there at least one (possible not minimal) repair for the given database with respect to the specified constraints? (ii) *repair checking*: given a set of update operations, is it a minimal repair? (iii) *consistent query answer*: is a given query true in every minimal repair?

- 3) We provide a method for computing *card-minimal* repairs for a database in presence of *steady aggregate constraints*, a restricted but expressive class of aggregate constraints. Under steady aggregate constraints, an instance of the problem of computing a *card-minimal* repair can be transformed into an instance of a Mixed-Integer Linear Programming (MILP) problem. Thus, standard techniques and optimizations addressing MILP problems can be re-used for computing a repairs.

On the basis of this data-repairing framework, we propose an architecture providing robust data acquisition facilities from input documents containing tabular data. We exploit integrity constraints defined on the input data to support the detection and the repair of inconsistencies in the data

arising from errors occurring in the acquisition phase performed on input data.

Organization

The thesis is organized as follows. Chapter 1 presents some preliminaries on first-order languages, relational databases, integrity constraints and queries. The definitions of repair and consistent query answer with some related computational problems are also introduced. Chapter 2 presents some proposal that were defined in the literature for the semantics of querying inconsistent integrated databases. In particular, we discuss the notion consistent query answer which has been used and extended in many subsequent works in literature. In Chapter 3, Chapter 4 and Chapter 5 we provide a comprehensive survey of the techniques for repairing and querying inconsistent databases. Specifically, Chapter 3 provides complexity results and techniques for the problem of computing consistent answers for several classes of queries and constraints. In Chapter 4 we discuss how repairs can be specified using logic programs with disjunction and classical negation, and consistent query answers can be obtained by skeptical reasoning. Both Chapter 3 and Chapter 4 refer to techniques where repairs are obtained by working at tuple-level, i.e. inserting and/or deleting whole tuples. In Chapter 5 we discuss the repairing techniques where repairs are obtained by working at attribute-level, i.e. updating attribute values.

In Chapter 6 we introduce *aggregate constraints* and characterize the problem of repairing and computing consistent answers in presence of such constraints under two different repair semantics (both defined for repairs working at attribute-level). Chapter 7 presents a specific but expressive form of aggregate constraints, namely *steady aggregate constraints*. An architecture based on a novel method for computing repairs is introduced. This architecture provides robust data acquisition facilities from input documents containing tabular data.

Finally, we summarize the work of the thesis and present some interesting research directions for future work.

Acknowledgements

It is a pleasure to express gratitude to all people which supported me, in any way, during the last three years I spent at D.E.I.S department of the University of Calabria. In particular, there are some people who deserve special thanks. I would like to thank my supervisor, Prof. Sergio Flesca, for guiding me with great enthusiasm during these years, and for his frank friendship. He has provided an excellent direction to my research activities and he has often inspired and incited me to investigate interesting issues. I express my gratitude

to Prof. Sergio Greco for valuable discussions and careful suggestions which have actually determined my course of doctoral study. I am also indebt with Filippo Furfaro who contributed with insightful proposals and comments to my education.

I say thank you to Prof. Ester Zumpano for discussions on some arguments of this thesis, and to Prof. Domenico Talia for his care in coordinating the PhD course I attended. Many thanks also to Giovanni Costabile and Francesco De Marte for cooperation and technical support. I thank my colleagues for their partnership in post-graduate studies and for their friendship. In particular, I thank Massimo Mazzeo, who have shared the office with me, for the recreation moments we spent together and for “personal communications”.

I am deeply grateful to my parents, Gaspare and Maria, and to my brothers, Antonio and Daniele, for their warm support and encouragement. Last but not least, I thanks my fiancée Alessandra who was carefully interested in my work, and who brought happiness to my life every time I was troubled.

Rende, November 2006

Francesco Parisi

Contents

Preface	iii
Main Contributions	viii
Organization	x
Acknowledgements	x
1 Preliminaries	1
1.1 First-Order Languages	1
1.2 Relational Databases	3
1.3 Integrity Constraints	3
1.3.1 Universal Integrity Constraints	4
1.3.2 Denial Constraints	6
1.3.3 Functional Dependencies	6
1.3.4 Inclusion Dependencies	7
1.4 Inconsistent Databases	7
1.5 Repairs	8
1.6 Queries	10
1.7 Consistent Query Answers	11
1.8 Computational Problems	12
1.9 Notations	13
2 Inconsistency in Databases: from Preliminary Approaches to Consistent Answers	15
2.1 Flexible Relational Model	16
2.1.1 Flexible Relational Algebra	18
2.2 Integrated Relational Calculus	22
2.2.1 Maximal Consistent Subset of a Relation	24
2.2.2 The Integrated Relational Model	25
2.2.3 Querying Integrated Relations	27
2.3 Merging Databases under Constraints	28
2.3.1 Semantics of Theory Merging	29
2.3.2 Result of Merging Databases under Constraints	30

2.4	Inconsistency in Databases as a Local Notion	33
2.4.1	Distinguishing between Consistent and Inconsistent Answers	34
2.5	Consistent Query Answers	35
2.5.1	Repairing by Inserting and Deleting a Minimal Set of Tuples	36
2.5.2	The Query Rewriting Approach	37
2.6	Discussion	42
3	The Tuple-Based Repairing Paradigm	45
3.1	Range-Consistent Query Answers	46
3.1.1	Conflict Graph	48
3.1.2	Complexity of Scalar Aggregation Queries	49
3.1.3	Other Tractable Cases	51
3.2	Repairing by Deleting a Minimal Set of Tuples	52
3.2.1	Denial Constraints	53
3.2.2	Inclusion Dependencies	57
3.3	Rewriting for a Class of Conjunctive Queries	59
3.3.1	The Class of Tree Queries	60
3.3.2	The Query Rewriting Algorithm	62
3.3.3	A Dichotomy Result	63
3.4	Rewriting SQL Queries	64
3.4.1	Join Queries	65
3.4.2	Aggregation Queries	67
3.5	A Class of Tractable but not Rewritable Queries	71
3.5.1	On the Class of Tractable Queries	73
3.6	A Large Perspective on Repair Semantics	74
3.6.1	Query Answering under Strict Repair Semantics	75
3.6.2	Query Answering under Loose Repair Semantics	77
3.7	Discussion	81
4	Logic Programs and Database Repairs	85
4.1	Logic Programs	85
4.1.1	General Logic Programs	85
4.1.2	Extended Logic Programs	86
4.1.3	Extended Disjunctive Logic Programs	87
4.1.4	Logic Programs with Exceptions	88
4.1.5	Prioritized Logic Programs	89
4.2	Querying Databases using Logic Programs with Exceptions	89
4.2.1	Extending Logic Programs with Exceptions	90
4.2.2	Specifying Repairs	90
4.2.3	Computing Consistent Query Answers	94
4.2.4	Referential Integrity Constraints	94
4.3	Querying Database using Extended Disjunctive Logic Programs	95
4.3.1	Computing Database Repairs	95

4.3.2	Computing Consistent Answers	98
4.3.3	Repair Constraints	100
4.3.4	Prioritized Repairs	102
4.4	Discussion	104
5	The Attribute-Based Repairing Paradigm	107
5.1	Repairing Census Data	108
5.1.1	Repairs for Census Data	109
5.1.2	Computing Repairs	112
5.2	Complexity and Approximation of Repairing Numerical Data	113
5.2.1	Least Square Repairs	113
5.2.2	Complexity Results and Approximations	115
5.3	An Heuristic for Repairing Inconsistent Databases	118
5.3.1	Minimum-Cost Repairs	118
5.3.2	A Greedy Algorithm Based on Equivalence Classes	121
5.4	Querying Inconsistent Databases by Means of Nuclei	124
5.4.1	Tableaux Formalism	126
5.4.2	Fixes and Repairs	128
5.4.3	Nuclei and Consistent Query Answers	131
5.5	Discussion	133
6	Repairing and Querying Numerical Databases under Aggregate Constraints	135
6.1	Introduction	135
6.2	Notations	138
6.3	Aggregate Constraints	138
6.4	Repairs	141
6.4.1	Reparability	143
6.4.2	Minimal Repairs	156
6.4.3	<i>Set</i> -Minimality versus <i>Card</i> -Minimality	159
6.5	Consistent Query Answers	161
6.6	Discussion	166
7	Computing Repairs for Inconsistent Numerical Data	169
7.1	Introduction	169
7.2	DART a Data Acquisition and Repairing Tool	172
7.3	Steady Aggregate Constraints	175
7.3.1	Complexity Results under Steady Aggregate Constraints	177
7.4	Computing a <i>Card</i> -Minimal Repair	178
7.5	DART Architecture	182
7.5.1	Acquisition Module	183
7.5.2	Data Extraction Module	184
7.5.3	Repairing Module	187
7.6	Discussion	188

Conclusions	191
References	195

Preliminaries

In this chapter we first present some preliminaries on first-order languages and relational databases. We assume that readers are familiar with first-order languages and relational databases and only recall here some definitions which will be used in this dissertation. Then, we introduce formal definitions of integrity constraints and queries providing a syntactic characterization of them.

After this, we provide the formal definition of repair for a (possible inconsistent) database with respect to a set of integrity constraints assuming that a repair semantics is given. Several forms of repair semantics have been proposed in the literature in the last few seven years. The characterization of the application context for which a given semantics is suitable will be discussed in the next chapters.

Finally, the notion of consistent query answers with some related computational problems will be introduced. These problems will be studied under different forms of repair semantics in the next chapters.

1.1 First-Order Languages

A first-order language \mathcal{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 . Predicate symbols are never 0-ary. We assume that the binary predicate symbol $=$ (equality relation) is defined. A 0-ary function symbol is called *constant*. A language \mathcal{L} is function-free if it only contains 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 e_1, \dots, e_n are terms.

The first order language \mathcal{L} over the alphabet Σ is defined as the set of all (*well-formed predicate calculus*) *formulas* that can be built using logical connectives (\neg, \wedge, \vee), quantifiers (\exists, \forall), terms and predicate symbols in the

standard way: $P(e_1, \dots, e_n)$ 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 $e_1 = e_2$ with e_1, e_2 terms; $\neg\varphi$, $\varphi \wedge \phi$, $\varphi \vee \phi$, $\exists x\varphi$ and $\forall x\varphi$ are formulas if φ, ϕ are formulas and x is a variable.

Free and *bound* occurrence of variables in formulas are recursively defined: 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 ; and this is extended to the other connectives. If ϕ is $\exists x\varphi$, then an occurrence of variable $y \neq x$ is free in ϕ if the corresponding occurrence 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 $\exists x$ at the beginning of ϕ .

A *sentence* is a well-formed formula that has no free variables 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 it involves no variables. A *literal* is an atom α or a negated atom $\neg\alpha$; in the former case, it is positive, and in the latter negative. Two literals are *complementary*, if they are of the form α and $\neg\alpha$, for some atom α .

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

The *Herbrand Universe* of a first-order language \mathcal{L} is the set of all ground terms that can be constructed using constant and function symbol of \mathcal{L} (if the language has no constants, then it is extended by adding an arbitrary new constant). The *Herbrand Base* of \mathcal{L} is the set of all ground atoms constructed from the predicates appearing in \mathcal{L} and the ground terms from *Herbrand Universe* as arguments. The Herbrand Universe and the Herbrand Base are both enumerable, and infinite if there is a predicate symbol of arity greater than zero. A *Herbrand interpretation* (or a *possible world*) 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 symbols 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 \mathcal{I} is a *model* of a set Φ of sentence, denoted as $\mathcal{I} \models \Phi$, if \mathcal{I} satisfies each formula in Φ . For a sentence Φ , an *Herbrand model* is a subset of the Herbrand Base satisfying Φ . The set of the models of Φ will be denoted as $Mod(\Phi)$. If $Mod(\Phi)$ is empty, then Φ is said to be *inconsistent* or *unsatisfiable*; otherwise it is said to be *consistent* or *satisfiable*.

We say that a sentence φ (logically) *implies* (or *supports*) ϕ , denoted as $\varphi \models \phi$, if $\text{Mod}(\varphi) \subseteq \text{Mod}(\phi)$, and φ is (logically) *equivalent* to ϕ , denoted as $\varphi \equiv \phi$, if $\text{Mod}(\varphi) = \text{Mod}(\phi)$.

A (first-order) *theory* is a set of sentence of the (first-order) language \mathcal{L} .

1.2 Relational Databases

We assume that there are finite (disjoint) sets of relation names **rel** and attribute names **att**. We also have a fixed, infinite database domain **dom**, consisting of uninterpreted constants, an infinite numeric domain \mathbb{Q} , consisting of all rational numbers, and an infinite numeric domain \mathbb{I} , consisting of all integer numbers. These domains are disjoint. We assume that elements of the domains with different names are different.

A *relation scheme* of a relation $P \in \mathbf{rel}$ is a sorted list (A_1, \dots, A_n) where $A_1, \dots, A_n \in \mathbf{att}$. A (*relational*) *database scheme* is a nonempty set of relation schemes. Each attribute A is typed and it has associated a domain denoted by $\text{DOM}(A)$. The *null* value \perp is not contained in $\text{DOM}(A)$ and $\text{DOM}_\perp(A) = \text{DOM}(A) \cup \perp$.

A *tuple* for a relation P is a mapping assigning to each attribute A of P an element in $\text{DOM}(A)$, i.e. it is a list of values $\langle a_1, \dots, a_n \rangle$ where a_i is the value of the attribute A_i , for each $i \in [1..n]$. The value a_i of the attribute A_i of a tuple t will be denoted as $t[A_i]$. For a set of attribute $\{A_i, \dots, A_j\}$, $t[A_i, \dots, A_j] = \langle t[A_i], \dots, t[A_j] \rangle$. A *relation instance* (or simply relation) is a set of tuples. In the following, a tuple $t = \langle a_1, \dots, a_n \rangle$ of a relation P , 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 .

Let \mathcal{L} be a function-free, first-order language with constant symbols in the domains **dom**, \mathbb{Q} and \mathbb{I} , and predicate symbols in **rel**. A *database instance* D can be seen as a finite Herbrand interpretation for \mathcal{L} . Since each instance is finite, it has finite active domain which is a subset of $\{\mathbf{dom} \cup \mathbb{Q} \cup \mathbb{I}\}$. We allow the standard built-in predicates $=, \neq, <, >, \leq, \geq$ over \mathbb{Q} and \mathbb{I} that have infinite, fixed extensions.

Given a database instance D , we will denote as $\text{Facts}(D)$ the set of ground atomic formulas $\{P(t) \mid D \models P(t)\}$, where P is a relation symbol and t a ground tuple.

1.3 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.

Definition 1.1 An *integrity constraint* is a the first-order sentence of the form:

$$(\forall X) [\Phi(X) \Rightarrow (\exists Z)\Psi(Y)] \quad (1.1)$$

where X, Y and Z are sets of variables, Φ and Ψ are two conjunctions of literals such that X and Y are the distinct set of variables appearing in Φ and Ψ , respectively, and $Z = Y - X$ is the set of variables existentially quantified. \square

In the definition above, the conjunction Φ is called the *body* and the conjunction Ψ the *head* of the integrity constraints. In both Φ and Ψ , one can find *relation literals* (i.e. either $P(w_1, \dots, w_n)$ or $\neg P(w_1, \dots, w_n)$ with P a relation symbol) and *built-in atoms* (comparison operators, e.g. $w = w'$ or $w \leq w'$).

The semantics of the above constraints is that for every value of X which makes the formula $\Phi(X)$ true there must be an instance of Z which makes $\Psi(Y)$ true.

Six common restrictions on the formula (1.1) give us six classes of integrity constraints:

1. The *full* (or *universal*) are those not containing existential quantified variables.
2. The *unirelational* (or *single-atom*) are those with one relation symbol only; dependencies with more than one relation symbols are called *multirelational*.
3. The *single-head* are those with a single atom in the head; dependencies with more than one atom in the head are called *multi-head*.
4. The *tuple-generating* are those in which no equality atoms occur.
5. The *equality-generating* are full, single-head, with an equality atom as head.
6. The *typed* are those whose variables are assigned to fixed positions of relation atoms and every equality atom involves a pair of variables assigned to the same position of the same relation atom; dependencies which are not typed will be called *untyped*.

Most of the dependencies developed in database theory are restricted cases of some of the above classes. For instance, functional dependencies are positive, unirelational, equality-generating constraints.

In this dissertation we deal with the following classes of integrity constraints.

1.3.1 Universal Integrity Constraints

Universal (single-head) integrity constraints are sentence of the form

$$\forall X [\beta_1 \wedge \dots \wedge \beta_n \wedge \neg\alpha_1 \wedge \dots \wedge \neg\alpha_m \wedge \varphi \Rightarrow \alpha_0]$$

where $\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_n$ are positive literals, φ is a conjunction of built-in atoms, α_0 is a positive atom or a built-in atom, X denotes the list of all variables appearing in β_1, \dots, β_n ; variables appearing in $\alpha_0, \dots, \alpha_m$, and φ also appear in β_1, \dots, β_n .

Often we will write constraints in a different format by moving literals from the head to the body and vice versa. For instance, the above constraint could be rewritten as

$$\forall X [\beta_1 \wedge \dots \wedge \beta_n \wedge \varphi \Rightarrow \alpha_0 \vee \alpha_1 \vee \dots \vee \alpha_m]$$

An universal integrity constraint is in *standard format* if it has the form

$$\forall X [\alpha_0 \vee \alpha_1 \vee \dots \vee \alpha_m \vee \neg\beta_1 \vee \dots \vee \neg\beta_n \vee \phi]$$

where the formula ϕ is equivalent to the negation of φ .

The positive literals α_i , or β_i , will be written explicitly as $P_i(X_i)$, where P_i is a relation symbol and X_i is a set of variables. Therefore we also write the universal constraints in *standard format* as

$$\forall X_0, \dots, X_n \left[\bigvee_{i=0}^m P_i(X_i) \vee \bigvee_{i=m+1}^n \neg P_i(X_i) \vee \phi(X_0, \dots, X_n) \right] \quad (1.2)$$

where P_0, \dots, P_n are relation symbols and X_0, \dots, X_n are tuples of variables and ϕ is a (quantifier-free) formula referring only to built-in predicates. Notice that there are no constants in the P_i ; if they are needed they can be pushed into ϕ .

Universal constraints that can be represented in standard format with $n \leq 2$ (they consists of two literals) will be called *binary constraints*.

In the following we will denote sets of universal (single-head) integrity constraints by *UC* and sets of binary constraints by *BC*.

Definition 1.2 Let *UC* be a set of universal integrity constraints in standard format over a database scheme \mathcal{D} . The set *UC* is said to be *acyclic* if there exists a function f from predicate names plus negations of predicate names in \mathcal{D} to the natural numbers, that is, $f : \{P_1 \dots, P_k, \neg P_1 \dots \neg P_k\} \rightarrow \mathbb{N}$, such that for every integrity constraint

$$\forall X_0, \dots, X_n \left[\bigvee_{i=0}^n L_i(X_i) \vee \phi(X_0, \dots, X_n) \right] \quad (1.3)$$

where L_i is a literal, and for every i and j ($0 \leq i, j \leq n$), if $i \neq j$, then $f(\neg L_i) > f(L_j)$. □

Here, $\neg L_i$ is the literal complementary to L_i , whereas f is a *level mapping*, similar to the mappings associated with stratified logic programs, except that complementary literals get values independently of each other.

Example 1.1 The set of universal integrity constraints

$$UC = \{\forall X [\neg P(X) \vee \neg Q(X) \vee S(X)], \forall XY [\neg Q(X) \vee \neg S(Y) \vee T(X, Y)]\}$$

is acyclic. The first constraint in UC implies $f(P) > f(S)$ and $f(Q) > f(S)$, whereas the second entails $f(Q) > f(T)$ and $f(S) > f(T)$. Therefore exists the function f defined by

$$\begin{array}{cccc} f(P) = 2 & f(Q) = 2 & f(S) = 1 & f(T) = 0 \\ f(\neg P) = 0 & f(\neg Q) = 0 & f(\neg S) = 1 & f(\neg T) = -2 \end{array}$$

which satisfies the condition of the definition above. □

1.3.2 Denial Constraints

Denial constraints are sentence of the form

$$\forall X_1, \dots, X_k \left[\bigvee_{i=1}^k \neg P_i(X_i) \vee \phi(X_1, \dots, X_k) \right] \quad (1.4)$$

where only negative literals appear in the standard format of the universal constraint. Often they will be written as

$$\forall X_1, \dots, X_k \neg [P(X_1) \wedge \dots \wedge P(X_k) \wedge \varphi(X_1, \dots, X_k)]$$

where φ is equivalent to the negation of ϕ . We will denote as DC sets of denial constraints.

In the following, the denial constraints with $k \leq 2$ will be called *binary denials*. Observe that binary denials are a subset of binary constraints. Moreover, any set of denial constraints is an *acyclic* (according to the Definition 1.2).

1.3.3 Functional Dependencies

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

$$\forall X_1, X_2, X_3, X_4, X_5 [P(X_1, X_2, X_4) \wedge P(X_1, X_3, X_5) \Rightarrow X_2 = X_3]$$

In the formula above, the expression $X_2 = X_3$ is the equality between the tuples of variables $X_2 = \langle y_1 \dots y_n \rangle$ and $X_3 = \langle z_1 \dots z_n \rangle$, that is $y_1 = z_1 \wedge \dots \wedge y_n = z_n$.

Often a functional dependency will be written as

$$\forall X_1, X_2, X_3, X_4, X_5 [\neg P(X_1, X_2, X_4) \vee \neg P(X_1, X_3, X_5) \vee X_2 = X_3]$$

A more familiar formulation of the above functional dependencies is $P[V] \rightarrow P[W]$ over a relation scheme P (or simply $V \rightarrow W$ if the relation scheme is

understood from the context), where V is the set of attributes of P corresponding to X_1 and W is the attribute of P corresponding to X_2 (and X_3).

Given a relation P with set of attributes U and set of functional dependencies FD over P , a *key* of P is a minimal (under \subseteq) set of attributes $K \subseteq U$ 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*.

The set of keys of a relation P will be denoted by $keys(P)$ and the primary key is denoted by $pkey(P)$. In the following we will denote by FD sets of functional dependencies, and by KD sets of key dependencies.

1.3.4 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 distinct sets of variables and $Y \subseteq X$; and P and Q relation symbols. They are often written as

$$\forall X \exists Z [\neg P(X) \vee Q(Y, Z)]$$

A more familiar formulation is $P[V] \subseteq Q[W]$, where V (resp. W) is the set of attributes of P (resp. Q) corresponding to Y .

Full (or universal) inclusion dependencies are those expressible without the existential quantifier. They are a special case of binary constraints. For instance, $\forall(X, Y)[P(X, Y) \Rightarrow Q(X)]$ is full.

Given two relations P and Q , and a key dependency $Q[K] \rightarrow Q[W]$, each inclusion dependency $P[V] \subseteq Q[K]$ is a *foreign key constraint*. If, additionally, K is the primary key of Q , then $P[V] \subseteq Q[K]$ is called *primary foreign key constraint*.

In the following we will denote by ID the sets of inclusion dependencies, and by FK sets of foreign key constraints.

Definition 1.3 Let ID be a set of inclusion dependencies over a database scheme \mathcal{D} . Consider a directed graph whose vertices are relations from \mathcal{D} and such that there is an edge $e(P, Q)$ in the graph iff there is an inclusion dependencies $P[V] \subseteq Q[W]$ in ID . A set of inclusion dependencies is *acyclic* if the above graph does not have a cycle. □

1.4 Inconsistent Databases

A database scheme contains knowledge on the structure of data, providing constraints on the form the data must have. The integrity constraints are usually

used for restricting the set of all possible instances that can be associated to a database scheme. They express relationships among data and prevent the insertion or deletion of data which could produce incorrect instances.

In the following, if not differently stated, we will assume that we are dealing with a *satisfiable* (or *consistent*) set of constraints, in the sense that there is a database instance that makes it true.

Definition 1.4 Given a database scheme \mathcal{D} and a set of integrity constraints IC on \mathcal{D} , an instance D of \mathcal{D} is said to be *consistent* w.r.t. IC if $D \models IC$ in the standard model-theoretic sense, *inconsistent* otherwise ($D \not\models IC$). \square

Example 1.2 Consider the database scheme \mathcal{D} consisting of only the relation $Student(Code, Name, Faculty)$, whose instance is shown in Fig. 1.1.

<i>Code</i>	<i>Name</i>	<i>Faculty</i>
<i>s</i> ₁	<i>Mary</i>	<i>Engeneering</i>
<i>s</i> ₂	<i>John</i>	<i>Science</i>
<i>s</i> ₂	<i>Frank</i>	<i>Engeneering</i>

Fig. 1.1. Database instance D

Assume that the functional dependencies $Code \rightarrow Name$ and $Code \rightarrow Faculty$ are defined for the database. The two functional dependencies can be expressed as follows

$$FD = \{ \forall x_1, x_2, x_3, x_4, x_5 [Student(x_1, x_2, x_4) \wedge Student(x_1, x_3, x_5) \Rightarrow x_2 = x_3] \\ \forall x_1, x_2, x_3, x_4, x_5 [Student(x_1, x_2, x_4) \wedge Student(x_1, x_3, x_5) \Rightarrow x_4 = x_5] \}$$

It is easy to see that $D \not\models IC$, since both the constraints in IC are violated: the second and the third tuple have the same value for $Code$, but different values for $Name$ and $Faculty$. \square

1.5 Repairs

A database instance D may be inconsistent w.r.t. a given set of integrity constraints IC . The restoration of the consistency in an inconsistent database can be achieved in a (possible infinite) number of ways, each of them yielding a consistent database. We have interest in minimal restorations of consistency, i.e. in performing actions that give us a new database instance R that shares the scheme with the original database D , but minimally differ from D according to some sort of *distance* between the original instance D and the

alternative consistent instance R . In literature the notion of minimal restoration of consistency for a database D has been captured in terms of *repair* for D . Different forms of semantics for repairs has been proposed. In this dissertation we will discuss these type of semantics relating them to the contexts where they are applicable.

Definition 1.5 Let D be a database instance of a database scheme \mathcal{D} and IC be a set of integrity constraints on \mathcal{D} . Let \mathcal{S} be a repair semantics. Given a partial order $\preceq_{\mathcal{S}}$ over databases instances (over \mathcal{D}) which depends on the repair semantics \mathcal{S} , a *repair* for D w.r.t. IC under \mathcal{S} is a new database instance R such that:

1. R is over the same scheme and domain as D ,
2. R satisfies IC ($R \models IC$),
3. there is no database instance R' such that $R' \preceq_{\mathcal{S}} R \preceq_{\mathcal{S}} D$, among the instances satisfying the first two conditions.

□

We will denote as $\mathcal{R}(D, IC, \mathcal{S})$ the set of all repairs for a database D w.r.t. the set of constraints IC under the semantics \mathcal{S} . When the semantics is understood we simply write $\mathcal{R}(D, IC)$.

The semantics \mathcal{S} determines the partial order $\preceq_{\mathcal{S}}$, which is defined over the set of consistent database instances for a given database D . Different type of actions can be performed on D for obtaining the restoration of consistency. We distinguish two *repairing paradigms*: *tuple-based* repairing paradigm and *attribute-based* repairing paradigm. In the former, only actions on *whole tuples* are allowed for restoring the consistency in a database. Whereas, in the latter also *value modifications* are allowed, that is a finer repair primitive consisting in correcting faulty values within the tuples is used.

Several repairing techniques that work according to tuple-based repairing paradigm are present in literature [4, 5, 7, 8, 9, 12, 23, 24, 27, 29, 44, 45, 46, 51, 52, 65, 66]. More recently, also techniques working at attribute level has been proposed [17, 42, 16, 41, 39, 76, 77, 78]. Several issues regarding the repair semantics and the two paradigms will be discussed in next chapters. Here, we only provide an example under the semantics that determines the order among consistent database instances on the basis of minimal-set (under \subseteq) of inserted and deleted tuples, which is the first semantics proposed in literature [4] (the technique in [4] will be discussed in Section 2.5).

The *distance* $\Delta(D, D')$ between database instances D and D' is the symmetric difference

$$\Delta(D, D') = \{Facts(D) - Facts(D')\} \cup \{Facts(D') - Facts(D)\}$$

The partial order $\preceq_{\mathcal{S}}$ among repairs of a database instance D according to the semantics \mathcal{S} equal to *minimal-set of inserted/deleted tuples* is defined by

$$R' \preceq_{\mathcal{S}} R'' \Leftrightarrow \Delta(D, R') \subseteq \Delta(D, R'')$$

Example 1.3 Consider the (inconsistent) database D and the set FD of constraints of Example 1.2. There are two different repairs R_1, R_2 for D w.r.t. FD , which are shown in Fig. 1.2. Both R_1 and R_2 are obtained deleting a (conflicting) tuple from the original instance D .

Code	Name	Faculty
s_1	Mary	Engeneering
s_2	John	Science

R_1

Code	Name	Faculty
s_1	Mary	Engeneering
s_2	Frank	Engeneering

R_2

Fig. 1.2. Repairs R_1 and R_2 for D

□

Observe that, according to this semantics, repairs may contain tuples that do not belong to the original database. For instance, removing violations of inclusion dependencies constraints can be done not only by deleting but also by inserting tuples.

Example 1.4 Consider the database scheme consisting of two unary relations P and Q . Assume that for an instance D , $Facts(D) = \{P(a), P(b), Q(a)\}$ and $IC = \{\forall x[P(x) \Rightarrow Q(x)]\}$. In this case we have two possible repairs for D w.r.t. IC . First, we can falsify $P(b)$, i.e. we delete a tuple and obtain the repair R_1 with $Facts(R_1) = \{P(a), Q(a)\}$. As second alternative, we can make $Q(b)$ true, i.e. we insert a tuple and obtain the repair R_2 with $Facts(R_2) = \{P(a), P(b), Q(a), Q(b)\}$.

□

The other dimensions of repairs, i.e. the granularity of the action performed (tuple-level or attribute-level), the type of the action (insertion, deletion, update) and other issue relative to the semantics (set-minimality, cardinality), will be deeply discussed in the next chapters.

1.6 Queries

Queries are (well-formed) formulas over the same language associated with the database scheme. A query is said to be *boolean* or a *sentence* (or *closed*) if it has no free variables. A sentence without (existential) quantifiers is called *quantifier-free* (or *universal*) sentence. A quantifier-free sentence without variables is called *ground* query.

A sentence is in *conjunctive normal form* (CNF) if it has the form $\Phi_1 \wedge \dots \wedge \Phi_m$ ($m \geq 1$), where each conjunct Φ_i has the form $L_1 \vee \dots \vee L_k$ ($k \geq 1$)

and where each L_j is a literal. Similarly, a sentence is in *disjunctive normal form* (DNF) if it has the form $\Phi_1 \vee \dots \vee \Phi_m$, where each disjunct Φ_i has the form $L_1 \wedge \dots \wedge L_k$ and where each L_j is a literal.

A query is in *prenex normal form* (PNF) if it has the form $q_1 x_1 \dots q_n x_n \Phi$, where each q_i is either \forall or \exists , and Φ is quantifier-free formula.

Conjunctive queries [26, 1] are formulas of the form

$$Q(w_1, \dots, w_m) = \exists z_1, \dots, z_k [P_1(u_1) \wedge \dots \wedge P_n(u_n) \wedge \varphi(u_1, \dots, u_n)]$$

where $w_1, \dots, w_m, z_1, \dots, z_k$ are all the variables that appear in the relation atoms $P_1(u_1) \dots P_n(u_n)$ of Q . We will say that w_1, \dots, w_m are the *free variables* of Q , whereas z_1, \dots, z_k are the existentially quantified variables of Q . Moreover, u_i are tuples of both variables and constants, and $\varphi(u_1, \dots, u_n)$ is a conjunction of built-in atomic formulas.

We will say that there is a *join* on a variable x if either (i) x appears in two literals $P_i(u_i)$ and $P_j(u_j)$ such that $i \neq j$, or (ii) there is a built-in equality atom $x = y$ and x appears in $P_i(u_i)$ and y appears in $P_j(u_j)$ such that $i \neq j$.

A conjunctive query is *simple* if (i) it has no repeated relation symbols, (ii) the variables in u_i are disjoint from that in u_j if $i \neq j$, and (iii) φ is of the form $\gamma_1(u_1) \wedge \dots \wedge \gamma_m(u_m)$ where γ_i is a built-in atomic formula.

Definition 1.6 A ground tuple $t = \langle a_1, \dots, a_m \rangle$ is an *answer* to a query $Q(w_1, \dots, w_m)$ in a database instance D if $D \models Q(t)$, i.e., the formula Q with $\langle w_1, \dots, w_m \rangle$ replaced by $\langle a_1, \dots, a_m \rangle$ is evaluated true in D . □

1.7 Consistent Query Answers

Given a (possible inconsistent) database D and a set of integrity constraints IC , the consistent answers to a query Q posed on D are those answers that are invariant under minimal form of restoration of the consistency of D w.r.t. IC [4], i.e. answers that can be obtained posing Q on every repair $R \in \mathcal{R}(D, IC, \mathcal{S})$, for a given semantics \mathcal{S} . From this perspective, the problem of computing consistent query answers is a form of caution reasoning from a database under integrity constraints.

Definition 1.7 (Consistent query answer) Let D be a database instance over the scheme \mathcal{D} , IC be a set of integrity constraints over \mathcal{D} and $Q(W)$ be a query over \mathcal{D} . Given a semantics \mathcal{S} , a ground tuple t is a *consistent answer* to Q w.r.t. IC under the semantics \mathcal{S} if for every repair $R \in \mathcal{R}(D, IC, \mathcal{S})$, it holds $R \models Q(t)$. □

In the following we will denote as $CQA(Q, D, IC, \mathcal{S})$ the set of consistent answer to Q in D w.r.t. IC under the semantics \mathcal{S} . If Q is a boolean query (a

sentence), then $CQA(Q, D, IC, \mathcal{S})$ is a singleton containing *true* if $R \models Q$ for every repair $R \in \mathcal{R}(D, IC, \mathcal{S})$, containing *false* otherwise. Additionally, Q is said to be *consistently true* in D if $CQA(Q, D, IC, \mathcal{S}) = \{true\}$, *consistently false* otherwise.

Example 1.5 Given the database D and the set of integrity constraints IC of Example 1.2, and the quantifier-free query $Q(x, y, z) = Student(x, y, z)$. The set of consistent answers to the query Q is $CQA(Q, D, IC, \mathcal{S}) = \{('s_1', 'Mary', 'Engineering')\}$, where \mathcal{S} the set-minimal semantics. \square

In the following, when \mathcal{S} is understood we simply write $CQA(Q, D, IC)$.

1.8 Computational Problems

In the following, if it is not differently stated, we will study the complexity of (decisional) problems adopting the *data complexity* assumption [1], which measures the complexity of a problem as a function of the size of a given database instance. The database scheme, the given query and integrity constraints are assumed to be fixed.

In this dissertation we will consider the following complexity classes [55, 68]:

- *P*TIME: the class of decision problems solvable in polynomial time by deterministic Turing Machines; this class is also denoted as P ;
- *NP*: the class of decision problems solvable in polynomial time by nondeterministic Turing Machines;
- *coNP*: the class of decision problems whose complements are solvable in *NP*;
- Σ_2^P : the class of decision problems solvable in polynomial time by nondeterministic Turing machines with an *NP* oracle; this class is also denoted as NP^{NP} ;
- Π_2^P : the class of decision problems whose complements are solvable in Σ_2^P ; this class is also denoted as $coNP^{NP}$;
- Δ_2^P : the class of decision problems solvable in polynomial time by deterministic Turing machines with an *NP* oracle; this class is also denoted as P^{NP} ;
- $\Delta_2^P[\log(n)]$: the class of decision problems solvable in polynomial time by deterministic Turing machines with an *NP* oracle which is invoked $\mathcal{O}(\log(n))$ times; this class is also denoted as $P^{NP[\log(n)]}$;
- AC^0 : the class of decision problems solvable by constant-depth, polynomial-size, unbounded fan-in circuits ($AC^0 \subset P$).

Let \mathcal{DC} be a class of databases instance over the same scheme \mathcal{D} , \mathcal{QC} be a class of queries \mathcal{QC} over \mathcal{D} , \mathcal{IC} be a class of integrity constraints over \mathcal{D}

and \mathcal{S} be a repair semantics. We study the (data) complexity of the following problems:

- *repair checking*, i.e. the complexity of determining the membership of the sets

$$\mathcal{RC}(IC, \mathcal{S}) = \{(D, R) \mid D, R \in \mathcal{DC} \wedge R \in \mathcal{R}(D, IC, \mathcal{S})\}$$

- *consistent query answers*, i.e. the complexity of determining the membership of the sets

$$\mathcal{CQA}(Q, IC, \mathcal{S}) = \{(D, t) \mid D \in \mathcal{DC} \wedge t \in \mathcal{CQA}(Q, D, IC, \mathcal{S})\}$$

where IC is a fixed finite set of integrity constraints belonging to \mathcal{IC} , Q is a fixed query belonging to \mathcal{QC} . If \mathcal{QC} is the class of Boolean Queries, t stands for *true*; otherwise t is a ground atom.

Given a class of complexity \mathcal{C} , the problem $\mathcal{RC}(IC, \mathcal{S})$ is \mathcal{C} -hard under the semantics \mathcal{S} if there is a finite set of integrity constraint \overline{IC} such that $\mathcal{RC}(\overline{IC}, \mathcal{S})$ is \mathcal{C} -hard. The problem $\mathcal{CQA}(Q, IC, \mathcal{S})$ is \mathcal{C} -hard (under the semantics \mathcal{S}) if there is a query $\overline{Q} \in \mathcal{QC}$ and a finite set of integrity constraints \overline{IC} such that $\mathcal{CQA}(\overline{Q}, \overline{IC}, \mathcal{S})$ is \mathcal{C} -hard.

In the following, when \mathcal{S} is understood we simply write $\mathcal{RC}(IC)$ and $\mathcal{CQA}(Q, D, IC)$.

1.9 Notations

In the following we generally use the following symbols (possibly with subscripts):

Constants: a, b, c

Variables: x, y, z

Set of variables: X, Y, Z

Terms: e, w

Tuples (of constants): $t, s, \langle a_1, \dots, a_n \rangle$

Tuples of both variables and constants: $u, v, \langle w_1, \dots, w_n \rangle$

Facts: $P(t), P(a_1, \dots, a_n)$

Atoms: $\alpha, \beta, P(u), P(w_1, \dots, w_n)$

Attributes: A, B, C

Set of attributes: K, U, V, W

Relation names (schemes): $P, Q, P(A, B, C), P(A_1, \dots, A_n)$

Database schemes: \mathcal{D}

Relation instances: I, J

Database instances: D

Inconsistency in Databases: from Preliminary Approaches to Consistent Answers

Often different databases are integrated together to provide a single unified view for the users. Generally, every source database is consistent with respect to a given set of integrity constraints, which is defined for entailing specific relationships among data of the (single) source. As result of database integration, many different kinds of discrepancies arise. In particular, possible discrepancies are due to (i) different sets of integrity constraints that are satisfied by different sources, and (ii) constraints that may be globally violated, even if every source database locally satisfies the same integrity constraints. Locating and resolving these conflicts may be difficult due to autonomy of the different databases. Moreover, when a query is posed on an inconsistent database, we can obtain some answers which are consistent with the constraints and others which are not.

Several proposals have been made in literature for the semantics of querying inconsistent integrated databases. In this chapter we will survey some selected approaches which addressed this issue.

In [2, 3] the *flexible relational model* and the *flexible algebra* have been proposed. They are, respectively, an augmentation of the relational model and an extension of the relational algebra. The *integrated relational model* [35] generalizes the flexible relational model, whereas the *integrated relational calculus* is a rather weak query language for integrated data. In [35] it is argued that the semantics of integrating (possible) inconsistent data is captured by the *maximal consistent subset* of the integrated data (although extended with *null* values). A different semantics, *merging by majority rule*, based on the *cardinality* of the source databases containing the same tuples, has been introduced in [62].

The notion of *consistent query answer* will be examined. It has been expressed in [20] as answers that are not involved in an integrity violation. The logical characterization of consistent answers, which has been used and extended in many subsequent works in literature, has been provided in [4]. An answer to query is consistent if it is the same answer which is obtained if the query would be posed to any minimally repaired version of the original database.

2.1 Flexible Relational Model

The *flexible relational model* extends classical relational model by supporting the representation of inconsistent data and also providing semantics for data manipulation operators in presence of potentially inconsistent data [3].

The problem of dealing with inconsistent data with respect to *primary key constraints* is addressed through the introduction of *flexible relations*, i.e. non 1NF relations that contain sets of non-key attributes. Moreover, *flexible algebra* which extends classical relational algebra is defined in order to provide semantics for database operation over flexible relations. The intent of these semantics is to perform meaningful operations in the presence of inconsistent data and also to provide as much information as possible to enable the user to resolve this inconsistency.

It is assumed that the set of constraints consists of only one primary key dependencies. Let $P(K, W)$ be a relation scheme, where K denotes the set of attributes in the primary key and W is the set of remaining attributes. P is said to be consistent if there is no pair of tuples t_1 and t_2 such that $\forall A \in K, t_1[A] = t_2[A]$ and $\exists B \in W$ with $t_1[B] \neq t_2[B]$.

In the *flexible relational model* data are described by means of *flexible relations*, which are derived from a (classical) relations by applying an operator *flexify* denoted by \sim . The flexible relation FR obtained by applying \sim to $P(K, W)$, denoted as $FR = \sim(P)$, has relation scheme $FR(K, W, Cons, Sel, Src)$, where *Cons* is the *consistent status attribute*, *Sel* is the *selection status attribute* and *Src* is the *source attribute*.

Thus, a flexible relation is derived from a classical relation by extending its scheme with the *ancillary* attributes and assigning values for these attributes for each of the tuples. Specifically, for each tuple t on the relation scheme $P(K, W)$, a tuple t' on the relation scheme $FR(K, W, Cons, Sel, Src)$ is derived as follows: $\forall A \in (K, W), t'[A] = t[A]$ and $t'[Cons] = true, t'[Sel] = true,$ and $t'[Src] = P$.

Example 2.1 Given the relation P on the left-hand side of the Fig. 2.1, the corresponding flexible relation FR is on the right-hand side of the figure.

K	A	B	C
10	X	⊥	Z
20	Y	⊥	V

P

K	A	B	C	$Cons$	Sel	Src
10	X	⊥	Z	true	true	P
20	Y	⊥	V	true	true	P

$FR = \sim(P)$

Fig. 2.1. Derivation of a flexible relation from a (classical) relation

□

A classical relation is consistent by definition and hence a flexible relation derived from a single classical relation is also consistent. Inconsistencies may arise if the integration of a set of consistent and autonomous databases is performed, i.e., when data from individually consistent flexible relations are merged. In order to represent inconsistent data in a flexible relation the notion of *Ctuple* is introduced.

A *Ctuple* on the scheme $FR(K, W, Cons, Sel, Src)$ is defined as a *cluster of tuples* having the same values for the key attributes, i.e., for any two tuples t_1 and t_2 from the *Ctuple* and for any attribute $A \in K$, $t_1[A] = t_2[A]$. A flexible relation is a set of *Ctuples*. Since the tuples of a *Ctuple* match on all the values of the attributes in K , in the context of a *Ctuple* the primary key for the tuples is the concatenation of the set of attributes K and the ancillary attribute Src . Each tuple in a given *Ctuple* has a unique value for the attribute Src and this value refers to the original source relation from which that tuple was derived. Depending on the tuples associated with a *Ctuple*, the *Ctuple* may be either consistent or inconsistent.

Let t_1 and t_2 be two tuples in the same *Ctuple* on the flexible relation scheme $FR(K, W, Cons, Sel, Src)$. Then, t_1 and t_2 are conflicting if there is some attribute $A \in W$ such that $t_1[A] \neq \perp$, $t_2[A] \neq \perp$ and $t_1[A] \neq t_2[A]$, where the interpretation given to the null value consists in *no information* [81] (a null value can be a place holder for either a nonexistent or an unknown value). A *Ctuple* is consistent if it contains non conflicting pairs of tuples, i.e. if for each attribute, all of the non-null values agree.

Observe that a *Ctuple* containing exactly a tuple is consistent by definition. Moreover, the ancillary attributes are not considered while determining the consistency between tuples. In fact, the value for the ancillary attribute $Cons$ is determined by evaluating the consistency of the tuples associated with a given *Ctuple*.

Example 2.2 Consider the three relations P_1 , P_2 and P_3 in Fig. 2.2 coming from the sources S_1 , S_2 and S_3 , respectively.

<table border="1" style="border-collapse: collapse; width: 80px; height: 40px;"> <thead> <tr><th style="padding: 2px;">K</th><th style="padding: 2px;">A</th><th style="padding: 2px;">B</th><th style="padding: 2px;">C</th></tr> </thead> <tbody> <tr><td style="padding: 2px;">10</td><td style="padding: 2px;">X</td><td style="padding: 2px;">⊥</td><td style="padding: 2px;">Z</td></tr> <tr><td style="padding: 2px;">20</td><td style="padding: 2px;">Y</td><td style="padding: 2px;">⊥</td><td style="padding: 2px;">Z</td></tr> </tbody> </table>	K	A	B	C	10	X	⊥	Z	20	Y	⊥	Z	<table border="1" style="border-collapse: collapse; width: 80px; height: 40px;"> <thead> <tr><th style="padding: 2px;">K</th><th style="padding: 2px;">A</th><th style="padding: 2px;">B</th><th style="padding: 2px;">C</th></tr> </thead> <tbody> <tr><td style="padding: 2px;">10</td><td style="padding: 2px;">X</td><td style="padding: 2px;">Y</td><td style="padding: 2px;">Z</td></tr> <tr><td style="padding: 2px;">20</td><td style="padding: 2px;">Y</td><td style="padding: 2px;">⊥</td><td style="padding: 2px;">Z</td></tr> </tbody> </table>	K	A	B	C	10	X	Y	Z	20	Y	⊥	Z	<table border="1" style="border-collapse: collapse; width: 80px; height: 40px;"> <thead> <tr><th style="padding: 2px;">K</th><th style="padding: 2px;">A</th><th style="padding: 2px;">B</th><th style="padding: 2px;">C</th></tr> </thead> <tbody> <tr><td style="padding: 2px;">10</td><td style="padding: 2px;">X</td><td style="padding: 2px;">W</td><td style="padding: 2px;">Z</td></tr> </tbody> </table>	K	A	B	C	10	X	W	Z
K	A	B	C																															
10	X	⊥	Z																															
20	Y	⊥	Z																															
K	A	B	C																															
10	X	Y	Z																															
20	Y	⊥	Z																															
K	A	B	C																															
10	X	W	Z																															
P_1	P_2	P_3																																

Fig. 2.2. Relations involved in the integration process

The flexible relation derived from the sources S_1 , S_2 and S_3 is as follows: It consists of two *Ctuples*: c_1 (containing the three tuples with key value 10) and c_2 (containing the last two tuples having key value 20). Note that the *Ctuple* c_2 is consistent whereas the *Ctuple* c_1 is not consistent.

K	A	B	C	$Cons$	Sel	Src
10	X	\perp	Z	<i>false</i>	<i>true</i>	S_1
10	X	Y	Z	<i>false</i>	<i>true</i>	S_2
10	X	W	Z	<i>false</i>	<i>true</i>	S_3
20	Y	\perp	Z	<i>true</i>	<i>true</i>	S_1
20	Y	\perp	Z	<i>true</i>	<i>true</i>	S_2

Fig. 2.3. The flexible relation obtained from integration of S_1 , S_2 and S_3

□

These ancillary attributes $Cons$, Sel and Src are instantiated by the application of the flexify operator. Each tuple of a flexible relation has a value for each ancillary attribute. The interpretation of these attributes is as follows.

- The $Cons$ attribute defines the consistency status of the $Ctuple$; its domain is $\{true, false\}$ and all tuples in the same $Ctuple$ have the same value.
- The Sel attribute denotes the selection status of the $Ctuples$ in the result of a selection operation performed on a flexible relation. It contains information about possible restrictions on the selection of tuples in $Ctuples$ and its domain is $\{true, false, maybe\}$; all tuples in the same $Ctuple$ have the same value. The value of this attribute is determined by applying the selection predicate (of flexible algebra) to data stored in each $Ctuples$. For flexible relations derived from source relations through the application of the *flexify* operator, its value is *true*, whereas for relations derived from other flexible relations its value can also be *false* or *maybe*.
- The Src attribute refers to the source relation from which a particular tuple has been derived. In case of inconsistent data, the source information is useful for determining the cause of a particular inconsistency and subsequently its resolution.

2.1.1 Flexible Relational Algebra

The *flexible algebra* defines a set of operation on the flexible relations, so that meaningful operation can be performed in the presence of conflicting data. The full algebra for flexible relation is defined in [2]; in this section, we briefly describe some of its operations. The set of $Ctuple$ operation includes *merging*, *equivalence*, *selection*, *union*, *cartesian product* and *projection*.

Merging

The merge operator merges the tuples in a $Ctuple$ in order to obtain a single nested tuple referred to as *merged Ctuple*.

Given a *Ctuple* c on the flexible relation scheme $FR(K, W, Cons, Sel, Src)$ containing the set of tuples $\{t_1, t_2, \dots, t_n\}$, the merged operator Ω applied to c returns a *merged Ctuple* $\Omega(c)$ such that

- i. for each $i \in [1..n]$ and $A \in \{K \cup \{Cons, Sel\}\}$, $\Omega(c)[A] = t_i[A]$;
- ii. for each $A \in \{W \cup \{Src\}\}$, $\Omega(c)[A] = \{\bigcup_{i \in [1..n]} t_i[A] - \{\perp\}\}$ if exists t_i such that $t_i[A] \neq \perp$, otherwise $\Omega(c)[A] = \perp$.

Thus, an attribute A of the *merged Ctuple* will be *null* (\perp) if and only if *null* is the unique value that A takes in the *Ctuple*.

Example 2.3 The merged relation derived from the relation shown in Fig. 2.3 is the following:

K	A	B	C	$Cons$	Sel	Src
10	X	$\{Y, W\}$	Z	<i>false</i>	<i>true</i>	$\{S_1, S_2, S_3\}$
20	Y	\perp	Z	<i>true</i>	<i>true</i>	$\{S_1, S_2\}$

Fig. 2.4. The merged relation

□

Equivalence

Two merged *Ctuples* $\Omega(c_1)$ and $\Omega(c_2)$ both on the scheme $FR(K, W, Cons, Sel, Src)$ are equivalent ($\Omega(c_1) \cong \Omega(c_2)$) if they do not conflict in any attribute, except for the *Src* attribute. More formally, $\Omega(c_1) \cong \Omega(c_2)$ if for each attribute $A \in \{K \cup \{Cons, Sel\}\}$, $\Omega(c_1)[A] = \Omega(c_2)[A]$, and for $A \in W$, the sets $\{\Omega(c_1)[A]\}$ and $\{\Omega(c_2)[A]\}$ coincide.

Observe that, the attribute *Src* is not considered while comparing *merged Ctuples*. Thus, two *merged Ctuples* are considered equivalent if they match exactly in each of their attributes other than the ancillary attribute *Src*.

Two *Ctuples* c_1 and c_2 are considered equivalent ($c_1 \equiv c_2$) if the corresponding *merged Ctuples* $\Omega(c_1)$ and $\Omega(c_2)$ are equivalent.

Given a flexible relation FR , a *Ctuple* c is member of FR , denoted $c \in FR$, if there is a *Ctuple* c' such that $c' \equiv c$ and c' is in FR .

Selection

The *Sel* attribute is modified after the application of selection operations. Specifically, for a given *Ctuple* c and a given selection predicate ϕ , the attribute *Sel* will be i) *true*, if ϕ is *true* for all tuples in c , ii) *false*, if ϕ is *false* for all tuples in c , and iii) *maybe* otherwise.

In classical relational algebra the selection operator determines the selection status of a tuple, i.e., *true* or *false*, for a given selection condition, which is specified by means of a selection predicate. In order to apply a selection predicate ϕ to a *Ctuple* c , ϕ is applied to the *merged Ctuple* $\Omega(c)$. Therefore, the semantics of the selection operator (in the flexible relational algebra) has to be extended to operate over non 1NF tuples, since a *merged Ctuple* is a nested tuple, i.e. its attributes may be associated with more than one value.

Given a flexible relation scheme $FR(K, W, Cons, Sel, Src)$ a *simple partial predicate* is of the form $(A \text{ op } \lambda)$ or $(A \text{ op } B)$ where $A, B \in K \cup W$, $op \in \{=, \neq, >, \geq, <, \leq\}$ and λ is a single value, i.e. $\lambda \in DOM(A) \cup \perp$.

Given a *Ctuple* c on the scheme $FR(K, W, Cons, Sel, Src)$, the predicate $(A \text{ op } B)$ evaluates to *true*, *false* or *maybe* as follows:

- *true*, if $\forall a_i \in \Omega(c)[A], \forall b_j \in \Omega(c)[B], (a_i \text{ op } b_j)$ is *true*.
- *false*, if $\forall a_i \in \Omega(c)[A], \forall b_j \in \Omega(c)[B], (a_i \text{ op } b_j)$ is *false*.
- *maybe*, otherwise.

The predicate $(A \text{ op } \lambda)$ is equivalent to $(A \text{ op } \{\lambda\})$. Moreover, since the semantics given to *null* is that of no information, any comparisons with *null* values evaluates to *false*.

A *partial selection predicate* is a conjunction of simple partial predicates. The status of a partial selection predicate is determined by the status of all its predicates according to the three-valued logic defined by the truth tables in Fig. 2.5. In these tables, *T* stands for *true*, *F* for *false* and *M* for *maybe*, whereas ϕ_1 and ϕ_2 refers to predicates.

ϕ_1	$\neg\phi_1$
<i>T</i>	<i>F</i>
<i>M</i>	<i>M</i>
<i>F</i>	<i>T</i>

$\neg\phi_1$

		ϕ_2		
		<i>T</i>	<i>M</i>	<i>F</i>
ϕ_1	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>
	<i>M</i>	<i>T</i>	<i>M</i>	<i>M</i>
	<i>F</i>	<i>T</i>	<i>M</i>	<i>F</i>

$\phi_1 \vee \phi_2$

		ϕ_2		
		<i>T</i>	<i>M</i>	<i>F</i>
ϕ_1	<i>T</i>	<i>T</i>	<i>M</i>	<i>F</i>
	<i>M</i>	<i>M</i>	<i>M</i>	<i>F</i>
	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>

$\phi_1 \wedge \phi_2$

Fig. 2.5. Truth tables for three-valued logic

When a partial predicate evaluates *maybe* represents the fact that different values are believed to be *true* by their respective sources for a particular attribute of a tuple. It is semantically quite different with respect to meaning that a finite set of values is available but it is not known which of the values is the real value.

Given a *Ctuple* c on the flexible relation scheme $FR(K, W, Cons, Sel, Src)$ and a partial selection predicate ϕ , the application of the selection operation $\sigma_\phi(c)$ return a *Ctuple* c' such that i) $\forall A \in \{K \cup W \cup \{Cons, Src\}\}, c'[A] = c[A]$, and ii) $c'[Sel] = \gamma_\phi(c) \wedge c[Sel]$, where $\gamma_\phi(c)$ is the result of applying ϕ to c .

Given a flexible relation FR , the result of the selection $\sigma_\phi(FR)$ is the set of $Ctuples$ $c \in FR$ such that either i) $\sigma_\phi(c') = true$ or $\sigma_\phi(c') = maybe$. Obviously, the $Ctuples$ having selection status $false$ are not present in the result.

Example 2.4 Considering the flexible relation $FR(K, A, B, C, Cons, Sel, Src)$ whose instance is shown in Fig. 2.3, the result of $\sigma_{(B=Y)}(FR)$ is the following

K	A	B	C	$Cons$	Sel	Src
10	X	⊥	Z	false	maybe	S_1
10	X	Y	Z	false	maybe	S_2
10	X	W	Z	false	maybe	S_3

Fig. 2.6. The flexible relation obtained by $\sigma_{(B=Y)}(FR)$

The selected $Ctuple$ have selection status *maybe* since the predicate is *false* for the tuples having key values $(10, S_1)$ and $(10, S_3)$, whereas it is *true* for the tuple having key values $(10, S_2)$, i.e. the attribute B is associated with conflicting data. □

Union

The union operator combines the tuples of two source $Ctuples$ in order to obtain a new $Ctuple$. This operation is meaningful if and only if the two $Ctuples$ represent data of the same concept, and so their scheme coincide and the value of the selection attribute is *true*. In the flexible algebra, the union operation has to be applied before any selection operation, because the selection operation can led to a loss of information.

Let c_1 and c_2 be two $Ctuples$ both on the flexible scheme $FR(K, W, Cons, Sel, Src)$ and such that $c_1[K] = c_2[K]$. The union of c_1 and c_2 , denoted as $c_1 \cup c_2$, is a $Ctuple$ c such that for each tuple $t \in c$, either $t \in c_1$ or $t \in c_2$.

The consistency of the resulting $Ctuple$ c , specified by the value of the attribute $Cons$, is evaluated after the union operation. Even if each of the source $Ctuples$ is independently consistent, it is still possible for the union of these $Ctuples$ to be inconsistent.

Given two flexible relation FR_1 and FR_2 , both on the scheme $FR(K, W, Cons, Sel, Src)$, the result of the union $FR_1 \cup FR_2$ is the set of $Ctuples$ c such that i) $c \in FR_1$ (resp. FR_2) and $\neg \exists c' \in FR_2$ (resp. FR_1) such that $c'[K] = c[K]$; ii) $c = c_1 \cup c_2$, where $c_1 \in FR_1$, $c_2 \in FR_2$ and $c_1[K] = c_2[K]$.

Example 2.5 The flexible relation FR shown in Fig. 2.3 is the union of relations $(\sim (P_1))$, $(\sim (P_2))$, and $(\sim (P_3))$, i.e. the flexible relations corresponding, respectively, to the (classical) relations P_1 , P_2 and P_3 , whose instances are shown in Fig. 2.2. \square

It is important to note that for flexible algebra $\sigma_\phi(FR_1 \cup FR_2)$ is not equivalent to $\sigma_\phi(FR_1) \cup \sigma_\phi(FR_2)$, i.e., in general, pushing selection over union of flexible relations entails an incorrect and incomplete result.

Example 2.6 As shown in Example 2.4, the selection $\sigma_{(B=Y)}(FR)$ results in the *Ctuple* in Fig. 2.6. Whereas $\sigma_{(B=Y)}(\sim (P_1)) \cup \sigma_{(B=Y)}(\sim (P_2)) \cup \sigma_{(B=Y)}(\sim (P_3))$ results in the *Ctuple* $\langle 10, X, Y, Z, true, true, S_2 \rangle$. \square

In [2] the issue related to optimization of flexible relation queries has been discussed and a strategies for pushing selection over unions has been proposed.

2.2 Integrated Relational Calculus

An extension of (classical) domain relational calculus, called *integrated relational calculus*, has been proposed in [35] to provide a logical semantics and a query language for manipulating data from autonomous multiple databases.

The integrated relational calculus is based on the definition of *maximal consistent subsets* for a possible inconsistent database. A *possible integration* of a set of relations is defined as the maximal consistent subset for the (possible) inconsistent relation obtained by union of these. This is achieved by means of extensions of relations by considering null values. Similarly to [3], null values denotes the absence of information [81]. It is assumed that tuples cannot have null values for the key attributes and that the values of the attributes in the primary key are correct, i.e. the consistency can not be obtained changing values of the primary key attributes.

The integrated relational calculus overcomes some drawbacks of the flexible relational algebra [3]. Specifically:

- i) the flexible relational algebra is not able to integrate possibly inconsistent relations if the associated relation scheme has more than one key;
- ii) the flexible relational model provides a rather weak query language.

The following two examples show two cases in which the flexible algebra fails as it is not able to detect the correct answer.

Example 2.7 Consider the database scheme containing the single binary relation $P_1(Employee, Wife)$ with two keys *Employee* and *Wife* with the former being the primary key. Assume there are the following two instances I_1 and I_2 for P_1 .

Employee	Wife
Terry	Lisa

I_1

Employee	Wife
Peter	Lisa

I_2

Fig. 2.7. Relation instance I_1 and I_2

Employee	Wife	Cons	Sel	Src
Terry	Lisa	true	true	I_1
Peter	Lisa	true	true	I_2

Fig. 2.8. The flexible relation obtained from I_1 and I_2

Integrating I_1 and I_2 using the flexible model we obtain the flexible relation $FR_1 = (\sim (I_1)) \cup (\sim (I_2))$ containing two non conflicting *Ctuples*, whose instance is the following.

Now asking “*Whose wife is Lisa?*” the flexible algebra will return the incorrect answer $\{Terry, Peter\}$, since $\sigma_{wife='Lisa'}(FR_1)$ returns FR_1 . In this example it is evident that flexible algebra fails in detecting the inconsistency in the data in I_1 and I_2 , due to the fact that *Wife* is a key. A correct answer would have been that it is undetermined who is the husband of *Lisa*. □

Example 2.8 Consider the relation $P_2(Employee, Department)$ with the attribute *Employee* as primary key. Assume there are the following two instances I_3 and I_4 for P_2 .

Employee	Department
Terry	CS

I_3

Employee	Department
Terry	Math

I_4

Fig. 2.9. Relation instance I_3 and I_4

By integrating I_3 and I_4 using the flexible model, the flexible relation FR_2 shown in Fig. 2.10 is obtained.

Now asking the question “*Who is employed in CS or Math?*”, represented by the selection formula $\sigma_{department='CS' \vee department='Math'}(FR_2)$, the expected answer is $\{Terry\}$, but flexible model will give \emptyset that is, it does not know who is working in *CS* or *Math*. Moreover in the flexible algebra there is no way to express a query like “*Who is possibly employed in Math ?*” □

<i>Employee</i>	<i>Department</i>	<i>Cons</i>	<i>Sel</i>	<i>Src</i>
<i>Terry</i>	<i>CS</i>	<i>false</i>	<i>true</i>	<i>R₁</i>
<i>Terry</i>	<i>Math</i>	<i>false</i>	<i>true</i>	<i>R₂</i>

FR₂**Fig. 2.10.** The flexible relation obtained from I_3 and I_4

2.2.1 Maximal Consistent Subset of a Relation

The model proposed by Dung in [35] generalizes the one of flexible relational algebra. It is argued that the semantics of integrating possibly inconsistent data is naturally captured by the *maximal consistent subsets* of the set of all information contained in the collection data.

Let $P(K, W)$ be a relation scheme where K is the set of attributes in the primary key and W the set of remaining attributes. Given two tuples t_1 and t_2 over P , they are said to be *conflicting* if there exists a key $K' \in \text{keys}(P)$ such that (i) for each $A \in K'$, $t_1[A] = t_2[A] \neq \perp$; and (ii) there is an attribute $B \in K \cup W$ such that $\perp \neq t_1[B] \neq t_2[B] \neq \perp$. Observe that, unlike [3], all keys of the relation are considered instead of only the primary key. A relation is consistent if there is no pair of conflicting tuples in it.

Interpreting *null* as no information leads to the following partial order on tuples. This order will be extended to relations to achieve the definition of maximal consistent subset of a (possible inconsistent) relation. Given two tuples t_1 and t_2 over $P(K, W)$, t_1 is said to be *less informative* than t_2 , denoted as $t_1 \ll t_2$ if and only if $\forall A \in K \cup W$, either $t_1[A] = \perp$ or $t_1[A] = t_2[A]$.

Two tuples t_1 and t_2 over the relation scheme $P(K, W)$ are said to be *joinable* if there is a tuple t' over $P(K, W)$, denoted as $\text{join}(t_1, t_2)$, such that $t_1 \ll t' \wedge t_2 \ll t'$. Since it is assumed that the value of each key attribute is not null, if t_1 and t_2 are joinable then $\forall A \in K$, $t_1[A] = t_2[A]$.

Given two joinable tuples t_1 and t_2 over the scheme $P(K, W)$, $(t_1 + t_2)$ denotes the tuple such that (i) $\forall A \in K$, $(t_1 + t_2)[A] = t_1[A] = t_2[A]$, and (ii) $\forall A \in W$, $(t_1 + t_2)[A] = \text{join}(t_1, t_2)[A]$. The tuple $(t_1 + t_2)$ represent the *sum* of the information in t_1 and t_2 .

Let I be a relation instance. The *informative closure* of I , denoted as \widehat{I} , is a relation obtained by adding to I the tuples t' such that either there is $t \in I$ and $t' \ll t$ or there are $t_1, t_2 \in I$ and $t' = (t_1 + t_2)$.

Example 2.9 Consider the relation scheme $P_3(\text{Employee}, \text{Tel}, \text{Salary})$ with $\text{key}(P_3) = \{\text{Employee}\}$ and the two instances I_5 and I_6 which are shown in Fig. 2.11.

The informative closures of I_5 and I_6 , i.e. \widehat{I}_5 and \widehat{I}_6 , are shown in Fig. 2.12.

In this case, only tuples which are less informative with respect to that already contained in the starting relations are added to the informative closure.

<i>Employee</i>	<i>Tel</i>	<i>Salary</i>
<i>Terry</i>	5709	35

I_5

<i>employee</i>	<i>tel</i>	<i>salary</i>
<i>Terry</i>	\perp	20

I_6

Fig. 2.11. Relation instances I_5 and I_6

<i>Employee</i>	<i>Tel</i>	<i>Salary</i>
<i>Terry</i>	5709	35
<i>Terry</i>	5709	\perp
<i>Terry</i>	\perp	35
<i>Terry</i>	\perp	\perp

\hat{I}_5

<i>Employee</i>	<i>Tel</i>	<i>Salary</i>
<i>Terry</i>	\perp	20
<i>Terry</i>	\perp	\perp

\hat{I}_6

Fig. 2.12. Informative closures of I_5 and I_6

But, as it will be clear in the following (see Example 2.10), when informative closures of union of relations is considered also tuples of type $(t_1 + t_2)$ will be added.

□

The notion of tuples less informative than others can be extended to relations, and it is exploited for defining the maximal consistent subset of a relation. Given two relation instances I_1 and I_2 over the relation scheme $P(K, W)$, I_1 is said to be *less informative* than I_2 ($I_1 \ll I_2$) if for each tuple $t_1 \in I_1$ there exists a tuple $t_2 \in I_2$ such that both $t_1[K] = t_2[K]$ and $t_1 \ll t_2$ hold.

Let I_1 be a relation instance. A *maximal consistent subset* of I_1 is a consistent relation instance I_2 over the same scheme of I_1 such that $I_2 \ll I_1$ and there is no consistent relation instance $I_3 \neq I_2$ such that $I_2 \ll I_3 \ll I_1$.

2.2.2 The Integrated Relational Model

The integration of data from autonomous databases is obtained by union of relation instances. If inconsistency arise from this step, the intent is that of obtaining more information with respect to considering only one database.

Let I_1, I_2 be two relation instances over the same relation scheme. If the information collected from I_1 and I_2 , represented by $J = I_1 \cup I_2$, is consistent then J represents the integration of information in I_1 and I_2 . Whereas, if $J = I_1 \cup I_2$ is inconsistent, a maximal consistent subset of the information contained in J would be one possible admissible collection of information that a user could extract from the integration.

Let I_1, \dots, I_n be relation instances over the same relation scheme $P(K, W)$.

- A *possible integration* of I_1, \dots, I_n is defined as a *maximal consistent subset* of \widehat{J} , where $J = \widehat{I}_1 \cup \dots \cup \widehat{I}_n$,
- The collection of all possible integrations of the instances I_1, \dots, I_n , denoted as $Integ(I_1, \dots, I_n)$, is defined as the semantics of integrating I_1, \dots, I_n .

Example 2.10 Consider the relations I_5 and I_6 , which are shown in Fig. 2.11, and their information closures \widehat{I}_5 and \widehat{I}_6 , which are shown in Fig. 2.12. Let J be the $\widehat{I}_5 \cup \widehat{I}_6$. The information closures \widehat{I} of J is shown in Fig. 2.13. Observe that the tuple $\langle Terry, 5709, 20 \rangle$ is added w.r.t. the tuples contained in $\widehat{I}_5 \cup \widehat{I}_6$.

Employee	Tel	Salary
Terry	5709	35
Terry	5709	⊥
Terry	⊥	35
Terry	⊥	⊥
Terry	⊥	20
Terry	⊥	⊥
Terry	5709	20

\widehat{J} with $J = \widehat{I}_5 \cup \widehat{I}_6$

Fig. 2.13. Informative closures of $\widehat{I}_5 \cup \widehat{I}_6$

There are two maximal consistent subsets S_1 and S_2 of \widehat{I} which are shown in Fig. 2.14. Therefore $Integ(I_5, I_6) = \{S_1, S_2\}$.

Employee	Tel	Salary
Terry	5709	35

S_1

Employee	Tel	Salary
Terry	5709	20

S_2

Fig. 2.14. Maximal consistent subsets of \widehat{J}

□

Example 2.11 Considering the relations I_1, I_2 in Fig. 2.7, it is not difficult to see that $Integ(I_1, I_2)$ consists of the relations S_3 and S_4 which are shown in Fig. 2.15. Whereas integration of I_3, I_4 , which are shown in Fig. 2.9, consists of the relations in Fig. 2.16.

□

<i>Employee</i>	<i>Wife</i>
<i>Terry</i>	<i>Lisa</i>
<i>Peter</i>	\perp

S_3

<i>Employee</i>	<i>Wife</i>
<i>Terry</i>	\perp
<i>Peter</i>	<i>Lisa</i>

S_4

Fig. 2.15. Integration of I_1 and I_2

<i>Employee</i>	<i>Department</i>
<i>Terry</i>	<i>CS</i>
<i>Terry</i>	\perp

S_5

<i>Employee</i>	<i>Department</i>
<i>Terry</i>	<i>Math</i>
<i>Terry</i>	\perp

S_6

Fig. 2.16. Integration of I_3 and I_4

2.2.3 Querying Integrated Relations

Queries over integrated data are formulated by means of a language derived by (classical) domain relational calculus, called *integrated relational calculus*. The extension consists in introducing a modal operator \mathcal{K} which allow us to “quantify” over the set of possible worlds, i.e. the collection of all possible integration of a set of relations.

Example 2.12 Consider the relation instances I_1 and I_2 , over the scheme $P_1(\textit{Employee}, \textit{Wife})$ of Example 2.7, and $\textit{Integ}(I_1, I_2)$ which is shown in Fig. 2.15. The query “*Whose wife is Lisa?*” can be formulated in the integrated relational calculus as:

- i) $Q_1(w) = \exists z [P_1(w, z) \wedge z = \textit{Lisa}']$ which can be stated as “*Whose wife is Lisa in a possible scenario?*”
- ii) $Q_2(w) = \exists z [\mathcal{K} (P_1(w, z) \wedge z = \textit{Lisa}')]$ which can be stated as “*Whose wife is Lisa in every scenario?*” Here the modal quantifier \mathcal{K} refers to all possible integrations, i.e. the set $\textit{Integ}(I_1, I_2)$ consisting of the relations S_3 and S_4 , which are shown in Fig. 2.15.

The answer to the query Q_1 is given by taking the union of the tuples matching the goal in all possible scenarios (brave reasoning), thus it consists of the set $\{\textit{Terry}, \textit{Peter}\}$. Whereas, the answer to the query Q_2 is obtained by considering the intersection of the tuples matching the goal in each possible scenario (cautious reasoning), thus in this case it is the empty set. □

Example 2.13 Consider the relation instances I_3 and I_4 , over the scheme $P_2(\textit{Employee}, \textit{Department})$ of Example 2.8, and $\textit{Integ}(I_3, I_4)$ which is shown

in Fig. 2.16. The query “*Who is possibly employed in Math?*” can be formulated in the integrated relational calculus as $Q_3(w) = \exists z [P_2(w, z) \wedge z = \text{'Math'}]$.

The question “*Who is employed in CS or Math (in every scenario)?*”, formulated as $Q_4(w) = \exists z [\mathcal{K}(P_2(w, z) \wedge (z = \text{'CS'} \vee z = \text{'Math'}))]$ gives the expected answer is $\{Terry\}$. □

In [35] the relationship between flexible relational algebra and integrated relational calculus was studied. It has been shown that flexible algebra is sound for the class of databases having exactly one key. Specifically, in the flexible relational model, the integration of relations I_1, \dots, I_n is defined as the union $I_1 \cup \dots \cup I_n$. If these relations are defined over a scheme with exactly a key, the set of relations obtained choosing exactly one tuple in every *Ctuple* of $I_1 \cup \dots \cup I_n$ corresponds to $Integ(I_1, \dots, I_n)$. For relations with exactly a key, expressions in flexible algebra can be transformed into equivalent formula of integrated relational calculus.

2.3 Merging Databases under Constraints

An approach for integrating conflicting information from different relational databases under constraints expressed as first-order sentences was proposed in [62]. The *merge* operator proposed extends the one in [63] to the first-order case and to deal with integrity constraints.

First, a formal semantics for merging multiple first-order theories under a set of constraints was proposed. The main properties of the approach is that it obtains maximal amount of information from each theory while observing the *majority rule* in case of conflict. Then, this semantics was applied to merge the information in databases under constraints, where a database is viewed as a first-order theory in which facts (but no rules) are involved.

Example 2.14 Suppose that three doctors, Doc_A , Doc_B and Doc_C , forming a committee are in consultation regarding two patients, $Jeff$ and Ed . There are three symptoms S_1 , S_2 and S_3 , and three possible diagnoses D_1 , D_2 and D_3 . Assume that $S_i(x)$ means that “*the patient x presents symptom S_i* ” and $D_i(x)$ means “*the patient x has disease D_i* ”. Suppose that the doctors Doc_A , Doc_B and Doc_C examine the patients independently and the their knowledge concerning the two patients is as follows:

$$\begin{aligned} Doc_A &= \{S_1(Jeff), S_2(Ed), \forall x S_1(x) \Rightarrow D_1(x)\} \\ Doc_B &= \{S_1(Jeff), \forall x S_1(x) \Rightarrow D_2(x), \forall x S_2(x) \Rightarrow D_3(x)\} \\ Doc_C &= \{S_3(Jeff), \forall x S_3(x) \Rightarrow D_1(x)\} \end{aligned}$$

It is known that “*no patient x has disease D_1 and D_2 at the same time*”, i.e. the integrity constraint $IC : \forall x D_1(x) \Rightarrow \neg D_2(x)$ holds.

When knowledge of doctors are merged, a conflict arise on the disease of *Jeff*, i.e. two doctors (Doc_A and Doc_C) diagnose $D_1(Jeff)$ and one (Doc_B) diagnoses $D_2(Jeff)$. Only a disease D_1 or D_2 can be diagnosed (since IC holds) and the committee will conclude $D_1(Jeff)$ following the *majority rule*.

Further, $D_3(Ed)$ will be concluded since the inconsistency about the disease of *Jeff* has no influence on the diagnosis of *Ed*. □

2.3.1 Semantics of Theory Merging

Let T_1, \dots, T_n be the (function free) first-order theories to be merged. Given the set of integrity constraints IC that must be satisfied by the merged theory, the result of merging is denoted as $Merge(\{T_1, \dots, T_n\}, IC)$.

The *models* of the resulting theory are those possible *worlds* (Herbrand interpretations) that are “closest” to the original theories, that is worlds which have the minimum *distance* from the theories $\{T_1, \dots, T_n\}$. In the following we first define the distance between a world w and a set of theories $\{T_1, \dots, T_n\}$ and then define the models which have *minimal* distance from $\{T_1, \dots, T_n\}$.

The distance between two possible world w and w' , denoted as $dist(w, w')$, is the *cardinality* of the symmetric difference of w and w' , that is

$$dist(w, w') = |(w - w') \cup (w' - w)|$$

The distance between a possible world w and a theory T is defined as

$$dist(w, T) = \min\{dist(w, w') \mid w' \in Mod(T)\}$$

where $Mod(T)$ denotes the set of all models of T . If $Mod(T) = \emptyset$, i.e. T is unsatisfiable, $dist(w, T) = 0$.

There may be possible worlds that are close to a particular theory but distant from others. The worlds that are closest overall to the set of theories $\{T_1, \dots, T_n\}$ are selected considering the *overall distance* between w and $\{T_1, \dots, T_n\}$, that is

$$dist(w, \{T_1, \dots, T_n\}) = \sum_{i=1}^n dist(w, T_i)$$

The models of the theory resulting of merging T_1, \dots, T_n with constraints IC are defined as follows:

$$Mod(Merge(\{T_1, \dots, T_n\}, IC)) = \{w \in Mod(IC) \mid dist(w, \{T_1, \dots, T_n\}) \text{ is minimum}\}$$

Thus, a possible world is a model of the theory $Merge(\{T_1, \dots, T_n\}, IC)$ if and only if it is a model of IC and its overall distance to the set of theories is minimum.

Example 2.15 Considering the Example 2.14, $Merge(\{Doc_A, Doc_B, Doc_C\}, IC)$ results in the following theory, from which $D_1(Jeff)$ and $D_3(Ed)$ are concluded:

$$\begin{aligned} & S_1(Jeff), S_3(Jeff), S_2(Ed) \\ & \forall x D_1(x) \Rightarrow \neg D_2(x) \\ & \forall x S_1(x) \Rightarrow D_1(x) \vee D_2(x) \\ & \forall x S_2(x) \Rightarrow D_3(x) \\ & \forall x S_3(x) \Rightarrow D_1(x) \end{aligned}$$

It is clear that replacing $\{S_1(Jeff), S_3(Jeff)\}$ with $\{S_2(Jeff)\}$ (and concluding $D_1(Jeff)$) a “less closed” theory than that would be obtained. \square

Consider the special case when $n = 1$, i.e. there is a single theory T_1 to be merged with constraints IC . Then if IC is viewed as new knowledge and T_1 as an old theory that must be revised, $Merge(\{T_1\}, IC)$ can be interpreted as a belief revision operation which incorporates a new sentence into an existing knowledge base. In the propositional case, this operator is equivalent to the revision operator of Dalal proposed in [34]. Dalal first used the Hamming distance as one kind of measurement of minimal change in belief revision.

2.3.2 Result of Merging Databases under Constraints

A database D is a first-order theory consisting of only a set ground atoms of the form $P(a_1, \dots, a_n)$, where P is a relation name and (a_1, \dots, a_n) are constants. Integrity constraints, expressed as first-order sentence, are requirements that the merged database must satisfy.

In the following we will show that for a set of databases $\{D_1, \dots, D_n\}$ and a set of integrity constraints IC , $Merge(\{D_1, \dots, D_n\}, IC)$ results in a disjunction of databases in conjunction with IC .

Let $D_1 \sqcup \dots \sqcup D_n$ be the *multiset* containing all the elements of the databases D_1, \dots, D_n . Let f be a function that removes duplicates from a multiset, i.e., for a given multiset M , $f(M)$ return a set containing all distinct elements in M . Then

$$Merge(\{D_1, \dots, D_n\}, IC) \equiv IC \wedge (D'_1 \vee \dots \vee D'_m),$$

where $D'_i = f(M_i)$ and M_i is a *maximum* (w.r.t. cardinality) *sub-multiset* of $D_1 \sqcup \dots \sqcup D_n$ such that $f(M_i) \wedge IC$ is consistent.

To compute the result of merging a set of databases $\{D_1, \dots, D_n\}$ under constraints IC , we first compute the sub-multisets M_1, \dots, M_m of $D_1 \sqcup \dots \sqcup D_n$, with each M_i having the maximal cardinality and such that they are consistent w.r.t. IC . If there is only one maximum sub-multiset, then the result is a single database obtained from the sub-multiset by applying f ; otherwise, the result

is a disjunction of databases, each transformed from one of the maximum sub-multisets. A disjunction of databases is obtained when there is no majority for a unique selection of a maximum sub-multiset, i.e. each of selections supported by a database is equally plausible.

Example 2.16 Consider the three database D_1 , D_2 and D_3 which are shown in Fig. 2.17, each containing a single relation $Bib(Author, Title, Year)$ collecting information regarding author, title and year of publication of papers. Assume that IC consists of the functional dependency

$$\forall x, y, z_1, z_2 [Bib(x, y, z_1) \wedge Bib(x, y, z_2) \Rightarrow z_1 = z_2]$$

Author	Title	Year
John	T1	1980
Mary	T2	1990

D_1

Author	Title	Year
John	T1	1981
Mary	T2	1990

D_2

Author	Title	Year
John	T1	1980
Frank	T3	1990

D_3

Fig. 2.17. Databases to be merged

Thus, $D_1 \sqcup D_2 \sqcup D_3$ is the multiset consisting of all the six tuples coming from D_1 , D_2 and D_3 . There is only one sub-multiset M of $D_1 \sqcup D_2 \sqcup D_3$ having the maximal cardinality 5 and such that it is consistent with IC . In a multiset the number of same elements make a difference in calculating cardinality. This is the key to achieving the majority principle in the merging process.

In Fig. 2.18 it is shown the multiset M and the resulting database obtained performing $f(M)$. The tuple $t_1 = \langle John, T1, 1981 \rangle$ does not belong to the result of merging. It is conflicting with the tuple $t_2 = \langle John, T1, 1980 \rangle$ and two database (D_1 and D_3) support t_2 whereas only one (D_2) supports t_1 . Thus the information that is maintained is t_2 since it is present in the majority of databases.

Author	Title	Year
John	T1	1980
John	T1	1980
Mary	T2	1990
Mary	T2	1990
Frank	T3	1990

M

Author	Title	Year
John	T1	1980
Mary	T2	1990
Frank	T3	1990

$f(M)$

Fig. 2.18. Maximal consistent sub-multiset and corresponding database

□

The number of resulting databases in the disjunction $D'_1 \vee \dots \vee D'_m$ can be (much) larger than n , the number of input databases. For a simple example, suppose $n = 1$ and $D_1 = \{P(a_1), P(a_2), \dots, P(a_m)\}$ and $IC = \neg P(a_1) \vee \neg P(a_2) \vee \dots \vee \neg P(a_m)$. Then the result is the disjunction of m databases, each of them containing $m - 1$ elements of D_1 .

Example 2.17 Assume that IC , D_1 and D_2 are as in Example 2.16, whereas D_3 is as shown in Fig. 2.19. In this case there are three consistent sub-multiset M_1, M_2, M_3 of $D_1 \sqcup D_2 \sqcup D_3$ which have the maximal cardinality 4. Therefore, three alternative databases can be obtained as result of merging, since there are no majority to resolve the conflict. They are succinctly represented in Fig. 2.20 by a single table where a nested tuple contains the set of alternative years $\{1980, 1981, 1982\}$ is used. Here the first tuple states that the year of publication of the book written by *John* with title $T1$ can be one of the values belonging to the set $\{1980, 1981, 1982\}$. For each of these values a different merged database is obtained.

<i>Author</i>	<i>Title</i>	<i>Year</i>
<i>John</i>	$T1$	1982
<i>Frank</i>	$T3$	1990

Fig. 2.19. Database D_3

<i>Author</i>	<i>Title</i>	<i>Year</i>
<i>John</i>	$T1$	$\{1980, 1981, 1982\}$
<i>Mary</i>	$T2$	1990
<i>Frank</i>	$T3$	1990

Fig. 2.20. Representation of the three databases resulting of merging

□

In the absence of integrity constraints the merge operation reduces to the union of the databases, i.e. $Merge(\{D_1, \dots, D_n\}, \emptyset) \equiv D_1 \cup \dots \cup D_n$. Whereas if IC is a set of *tuple generating dependencies* then $Merge(\{D_1, \dots, D_n\}, IC) \equiv D_1 \cup \dots \cup D_n \cup IC$.

The discussed approach for integrating conflicting information from different databases with uniform scheme is also extended in [62] to merging databases with conflicting schemes under constraints. The basic idea is that

of adding scheme transformation rules to constraints that the merged database must satisfy.

2.4 Inconsistency in Databases as a Local Notion

The notion of consistent query answer in inconsistent databases has been considered for the first time in [20]. The proposed approach consists in distinguishing two kind of answers, *consistent answers* that do not depend on data involved in an integrity constraint violations, and *inconsistent answers* that are derived from data violating some integrity constraints.

It was argued that classical logic is inappropriate for the formalization of information systems because of its *global* notion of inconsistency. Then it has been shown that *minimal logic* provides a formalism where inconsistency can be seen as a *local* notion. Consistent query answers has been defined on the basis of provability in minimal logic.

If some integrity constraints are violated in a databases then its logical specification results inconsistent. Clearly, there are no models for an inconsistent specification and every possible expression (in the language) is an answer for a query Q .

Example 2.18 Consider a relational database consisting of a single relation $P(\textit{Employee}, \textit{Age}, \textit{Salary})$ whose instance is shown in Fig. 2.21. Assume that the integrity constraint stating that there is no employee with age less than 18 is defined, i.e. $\forall x, y, z [\neg P(x, y, z) \vee y \leq 18]$.

<i>employee</i>	<i>age</i>	<i>salary</i>
<i>Jack</i>	25	1000
<i>Mary</i>	15	1100
<i>Frank</i>	20	1000

Fig. 2.21. Relation P

Consider the boolean query $Q = \exists x P(\textit{Jack}, x, 500000)$, asking if the salary of employee *Jack* is 500000. Since in classical logic every formula can be derived from inconsistent specification, then the answer to the query is *true*. \square

The idea expressed in [20] is that it is acceptable that data violating a given set of integrity constraints corrupt *some* answers, but not that they affect *all* answers.

Example 2.19 Considering the Example 2.18, since the salary of *Jack* is not related to the (inconsistent) age of *Mary* one expects that the answer to the query $Q = \exists x P(\textit{Jack}, x, 500000)$ is *false*.

□

Minimal logic, a weakening of classical logic, was proposed as basis for information system formalization and for query answering. When minimal logic is considered instead of classical logic we lost:

1. proofs based on an elimination of double negations, i.e. proofs of an atom α from a theory T resulting from a derivation of $\neg\neg\alpha$, and
2. refutations proofs, i.e. proofs of an atom α from a theory T resulting from a derivation of *false* from $T \cup \{\alpha\}$.

The *ex falso quodlibet* principle does not hold in minimal logic, i.e. it is no possible to derive every formula in an inconsistent theory. Inconsistency in minimal logic appear as *local* in the sense that they do not give arise to the derivation of every possible formula.

In [20] it has been shown that minimal logic is incomplete for full first-order logic, whereas completeness for *positive* (definite or disjunctive) deductive databases *without* integrity constraints was shown in [19]. Basically, this means that for positive deductive databases without integrity constraints, every answer which can be computed within classical logic can also be computed within minimal logic.

2.4.1 Distinguishing between Consistent and Inconsistent Answers

The approach to query answering that has been proposed exploits the local notion of inconsistency in minimal logic. It makes possible to recognize whether the answer to a query has been derived from possibly corrupted data.

Let denote by \vdash_m the derivability in minimal logic. Given a database D and a set of integrity constraints IC , an *inconsistency kernel* of D w.r.t. IC , denoted as $Kernel(D, IC)$, is defined as a minimal (under \subseteq) subset of D such that $(Kernel(D, IC) \cup IC) \vdash_m \text{false}$, that is $Kernel(D, IC)$ contains at least a violation of an integrity constraint in IC .

Given a database D and a set of integrity constraints IC , a closed formula Φ is said to be an *inconsistent answer* of D w.r.t. IC if

1. Φ is derivable in minimal logic from D ($D \vdash_m \Phi$), and
2. for every subset $S \subseteq D$ such that $S \vdash_m \Phi$, there is an inconsistent kernel $Kernel(D, IC)$ such that $(S \cap Kernel(D, IC)) \neq \emptyset$.

Informally, an inconsistent answer is an answer which can not be established without making use of some data involved in a derivation of an integrity constraint violation.

Given a database D and a set of integrity constraints IC , a closed formula Φ is said to be an *consistent answer* of D w.r.t. IC if

1. there is a subset $S \subseteq D$ such that $S \vdash_m \Phi$, and
2. for every inconsistent $Kernel(D, IC)$, $(S \cap Kernel(D, IC)) = \emptyset$.

Example 2.20 Consider a relational database D of Example 2.18 and the integrity constraint $\forall x, y, z [\neg P(x, y, z) \vee y \leq 18]$. The inconsistent kernel $Kernel(D, IC)$ is $\{P(Mary, 15, 1100)\}$, therefore the answer *false* to the query $Q = \exists x P(Jack, x, 500000)$ is a consistent answer, whereas the answer *true* to the query $Q = \exists x P(Mary, x, 1100)$ is an inconsistent answer. \square

Incidentally, we note that the definition of consistent answers above also holds for positive disjunctive deductive databases, because of the minimal logic is complete for *positive* definite or disjunctive deductive databases *without* integrity constraints [19].

In [20] the consistent query answers based on provability in minimal logic was defined, but no computational mechanism for obtaining such answers was given.

2.5 Consistent Query Answers

The technique proposed in [4] introduces a logical characterization of *consistent query answers* in relational databases that may be inconsistent with the given integrity constraints. An answer to query posed to a database that violates the integrity constraints is consistent if it is the same answer which is obtained if the query would posed to any minimally repaired version of the original database.

A method for computing such answers was provided. Basically it consist of a query rewriting which preserve the original database instance. That is, on the basis of a query Q , the method computes (using an iterative procedure) a new query $T_\omega(Q)$ whose evaluation in a (consistent or inconsistent) database returns the set of consistent answers to the original query Q . Intuitively, the transformed query $T_\omega(Q)$ is qualified with appropriate information derived from the interaction between the query Q and the integrity constraints. This forces the (local) satisfaction of the integrity constraints and makes it possible to discriminate between the tuples in the answer set. The method is inspired by *semantic query optimization* [25], where the notion of *residue* was developed in the context of deductive databases for optimizing the process of answering queries using the semantics knowledge about the domain that is contained in the integrity constraints.

The constraints considered are *universal integrity constraints* expressed in *standard format* (cfr. Section 1.3.1). Functional dependencies, *full* inclusion dependencies, and transitivity constraints of the form $\forall X, Y, Z [P(X, Y) \wedge P(Y, Z) \Rightarrow P(X, Z)]$ are examples of integrity constraints that can be transformed to the standard format. Whereas, an inclusion dependency of the form $\forall X [P(X) \Rightarrow \exists Y Q(X, Y)]$ cannot be transformed to the standard format.

2.5.1 Repairing by Inserting and Deleting a Minimal Set of Tuples

The notion of *repair*, anticipated in Section 1.5 under a generic semantics, has been firstly introduced in [4] with the semantics of *minimal-set of whole tuples*. Under this semantics the distance between database instances D and D' is defined as the symmetric difference of $Facts(D)$ and $Facts(D')$, i.e. $\Delta(D, D') = \{Facts(D) - Facts(D')\} \cup \{Facts(D') - Facts(D)\}$.

A database instance R is a *repair* of a database instance D w.r.t. a set of integrity constraints IC if

1. R is over the same scheme and domain as D ,
2. R satisfies IC ,
3. the distance $\Delta(D, R)$ is minimal under set containment among the instances satisfying the first two conditions.

Thus the atomic primitive used for providing a new consistent instance R from a (possible inconsistent) database D is based on inserting/deleting whole tuples.

Example 2.21 Consider the database D_1 consisting of the instance of the relation $Student(Code, Name, Faculty)$, which is shown in Fig. 2.22.

<i>Code</i>	<i>Name</i>	<i>Faculty</i>
<i>s₁</i>	<i>Mary</i>	<i>Engeneering</i>
<i>s₂</i>	<i>John</i>	<i>Science</i>
<i>s₂</i>	<i>Frank</i>	<i>Engeneering</i>

Fig. 2.22. Database instance D_1

Assume that the following set FD of the functional dependencies (expressed in *standard format*) is defined for D_1 .

$$FD = \{ \forall x_1, x_2, x_3, x_4, x_5 [\neg Student(x_1, x_2, x_4) \vee \neg Student(x_1, x_3, x_5) \vee x_2 = x_3] \\ \forall x_1, x_2, x_3, x_4, x_5 [\neg Student(x_1, x_2, x_4) \vee \neg Student(x_1, x_3, x_5) \vee x_4 = x_5] \}$$

The database D_1 is inconsistent w.r.t. FD , since there are two tuples with the same key. There are two different repairs R_1, R_2 for D_1 w.r.t. FD , which are shown in Fig. 2.23. Both R_1 and R_2 are obtained deleting a (conflicting) tuple from the original instance D_1 . □

According to this semantics, repairs may contain tuples that do not belong to the original database. For instance, removing violations of inclusion dependencies constraints can be done not only by deleting but also by inserting tuples.

Code	Name	Faculty
s_1	Mary	Engeneering
s_2	John	Science

R_1

Code	Name	Faculty
s_1	Mary	Engeneering
s_2	Frank	Engeneering

R_2

Fig. 2.23. Repairs R_1 and R_2 for D_1

Example 2.22 Consider the database scheme consisting of two unary relations P and Q . Assume that for an instance D_1 , $Facts(D_1) = \{P(a), P(b), Q(a)\}$ and $ID = \{\forall x[P(x) \Rightarrow Q(x)]\}$, whose standard format is $ID = \{\forall x[\neg P(x) \vee Q(x)]\}$. In this case we have two possible repairs for D_1 w.r.t. ID . First, we can falsify $P(b)$, i.e. we delete a tuple and obtain the repair R_1 with $Facts(R_1) = \{P(a), Q(a)\}$. As second alternative, we can make $Q(b)$ true, i.e. we insert a tuple and obtain the repair R_2 with $Facts(R_2) = \{P(a), P(b), Q(a), Q(b)\}$. \square

The notion of *consistent query answer* for a (possible inconsistent) database D w.r.t. a set of integrity constraints IC (cfr. Section 1.7) has been introduced in [4]. It has been formalized as the set of answers that are returned in all repairs for D .

Example 2.23 Considering the database D_1 and the constraints FD of Example 2.21, the consistent answers to the query $Q_1(x, y, z) = Student(x, y, z)$ is (' s_1 ', 'Mary', 'Engeneering').

Further, the query $Q_2 = \exists x P(x)$ is consistently true in the database D_2 of Example 2.22, since the ground atom $P(a)$ belongs to the repairs for D_2 w.r.t. ID . \square

2.5.2 The Query Rewriting Approach

The technique is based on the computation of an equivalent query $T_\omega(Q)$ derived from the source query Q . It is expected that for every query Q , database D and set of constraints IC , the set of consistent answers to Q in D w.r.t. IC , $CQA(Q, D, IC)$, is exactly the set of ground tuples t obtained posing the rewritten query $T_\omega(Q)$ on the original database D , i.e. the set of tuples $\{t \mid D \models T_\omega(Q(t))\}$.

Now we present the technique and then we discuss the conditions (classes of query and constraints) which ensure the soundness and completeness of the technique.

The form of the queries Q for which the query-rewriting operator $T_\omega(Q)$ is defined are first order formulas. They are assumed to be in *prefix disjunctive normal form*, that is, having the following syntactical form

$$\overline{Q} \bigvee_{i=1}^s \left(\bigwedge_{j=1}^{m_i} P_{i,j}(u_{i,j}) \wedge \bigwedge_{j=m_i+1}^{n_i} \neg P_{i,j}(v_{i,j}) \wedge \Psi_i \right) \quad (2.1)$$

where \overline{Q} is a sequence of quantifiers and every Ψ_i contains only built-in predicates.

The definition of $T_\omega(Q)$ is based on the notion of *residue* developed in the context of *semantics query optimization*. Specifically, given a query Q , a new query $T_\omega(Q)$ is computed by iterating the operator T which transforms the query by conjoining the corresponding residues to each database literal appearing in the query Q , until a fixed point is reached. The residues of a database literal force the satisfaction of the integrity constraints for the tuples satisfying the literal and are obtained by resolving the literal with the integrity constraints.

Example 2.24 Consider the integrity constraint $\forall x[P_1(x) \vee \neg P_2(x) \vee \neg P_3(x)]$. If the query is $Q = P_3(x)$ then $\forall x(P_1(x) \vee \neg P_2(x))$ must be true, otherwise the constraint would be violated. The universal quantified first order formula $\forall x(P_1(x) \vee \neg P_2(x))$ is the residue for $P_3(x)$. In order to obtain the consistent query answers to Q , the query $T_\omega(Q) = P_3(x) \wedge \forall x[P_1(x) \vee \neg P_2(x)]$ will be posed on the database. \square

Residues are generated as follows. Given an integrity constraint ic in *standard format*

$$ic = \forall X_0, \dots, X_n \left[\bigvee_{i=0}^m P_i(X_i) \vee \bigvee_{i=m+1}^n \neg P_i(X_i) \vee \phi(X_0, \dots, X_n) \right]$$

the residues for a negative literal $L_n = \neg P_j(X_j)$ and a positive literal $L_p = P_j(X_j)$, both w.r.t. ic , are respectively:

$$Res(L_n, ic) = \overline{Q} \left(\bigvee_{i=1}^{j-1} P_i(X_i) \vee \bigvee_{i=j+1}^m P_i(X_i) \vee \bigvee_{i=m+1}^n \neg P_i(X_i) \vee \phi(X_1, \dots, X_n) \right)$$

$$Res(L_p, ic) = \overline{Q} \left(\bigvee_{i=1}^m P_i(X_i) \vee \bigvee_{i=m+1}^{j-1} \neg P_i(X_i) \vee \bigvee_{i=j+1}^n \neg P_i(X_i) \vee \phi(X_1, \dots, X_n) \right)$$

where \overline{Q} is a sequence of universal quantifiers over all the variables not appearing in X_j .

If $\{\rho_1, \dots, \rho_r\}$ is the (possible empty) set of residues for a literal $L(X_j)$, i.e. $\bigcup_{ic \in IC} Res(L, ic) = \{\rho_1, \dots, \rho_r\}$, then the rule $L(Y) \mapsto L(Y)\{\rho_1(Y), \dots, \rho_r(Y)\}$ is generated, where Y is a set of new variables.

Example 2.25 Considering the query $Q(x, y, z) = Student(x, y, z)$ of Example 2.23 and the set constraints $IC = \{ic_1, ic_2\}$ introduced in the Example 2.21, where:

$$ic_1 = \forall x_1, x_2, x_3, x_4, x_5 [\neg Student(x_1, x_2, x_4) \vee \neg Student(x_1, x_3, x_5) \vee x_2 = x_3]$$

$$ic_2 = \forall x_1, x_2, x_3, x_4, x_5 [\neg Student(x_1, x_2, x_4) \vee \neg Student(x_1, x_3, x_5) \vee x_4 = x_5]$$

The residue obtained by resolving Q with the former constraint is

$$Res(Student(x_1, x_2, x_4), ic_1) = \forall x_3, x_5 [\neg Student(x_1, x_3, x_5) \vee x_2 = x_3]$$

whereas by resolving Q with the latter constraint the residue is the following

$$Res(Student(x_1, x_2, x_4), ic_2) = \forall x_3, x_5 [\neg Student(x_1, x_3, x_5) \vee x_4 = x_5]$$

Thus, the following rule is generated

$$Student(y_1, y_2, y_3) \mapsto Student(y_1, y_2, y_3) \{ \forall y_4, y_5 [\neg Student(y_1, y_4, y_5) \vee y_2 = y_4], \\ \forall y_4, y_5 [\neg Student(y_1, y_4, y_5) \vee y_3 = y_5] \}$$

In order to obtain the consistent query answer to Q , the query Q is rewritten as follows

$$T_\omega(Q) = Student(y_1, y_2, y_3) \wedge \forall y_4, y_5 [\neg Student(y_1, y_4, y_5) \vee y_2 = y_4] \wedge \\ \forall y_4, y_5 [\neg Student(y_1, y_4, y_5) \vee y_3 = y_5]$$

□

In general, depending on the integrity constraints and the original query, we may need to consider the residues of residues and so on. Therefore a family of operator T_n , with $n \geq 0$, and T_ω is considered.

The application of an operator T_n to a query Q is defined inductively by means of the following rules

1. $T_0(Q) = Q$;
2. for each (positive or negative) literal $L(Y)$, if there is the rule $L(Y) \mapsto L(Y) \{ \rho_1(Y), \dots, \rho_r(Y) \}$, then $T_n(L(Y)) = L(Y) \wedge \bigwedge_{i=1}^r T_{n-1}(\rho_i(Y))$;
3. if Q is the formula in prenex disjunctive normal form as in (2.1), then, for every $n \geq 0$,

$$T_n(Q) = \overline{Q} \bigvee_{i=1}^s \left(\bigwedge_{j=1}^{m_i} T_n(P_{i,j}(u_{i,j})) \wedge \bigwedge_{j=m_i+1}^{n_i} T_n(\neg P_{i,j}(v_{i,j})) \wedge \Psi_i \right)$$

The application of the operator T_ω on a query Q is defined as

$$T_\omega(Q) = \bigcup_{n < \omega} \{T_n(Q)\} = \{Q_1, \dots, Q_m\}$$

Given a database D , a (ground) tuple t is an *answer* to a set of queries $\{Q_1, \dots, Q_m\}$ if $D \models Q_1(t) \wedge \dots \wedge Q_m(t)$.

Supplier	Department	Item
s_1	d_1	i_1
s_2	d_2	i_2

Item	Type
i_1	t_1
i_2	t_1

Supply
Class

Fig. 2.24. Relations *Supply* and *Class*

Example 2.26 Consider the database D consisting of the relations *Supply* and *Class*, whose instances are shown in Fig. 2.24.

Assume that the following integrity constraint is defined, stating that s_1 is the only supplier of items of class t_1 :

$$\forall x_1, x_2, x_3 [Supply(x_1, x_2, x_3) \wedge Class(x_3, t_1) \Rightarrow x_1 = s_1]$$

The database D is inconsistent (an item of type t_1 is also supplied by supplier s_2).

The constraint can be rewritten in the standard format as follows (the constant t_1 is pushed into the built-in predicate formula):

$$ic = \forall x_1, x_2, x_3, x_4 [\neg Supply(x_1, x_2, x_3) \vee \neg Class(x_3, x_4) \vee x_4 \neq t_1 \vee x_1 = s_1]$$

The residue of the literals appearing in ic are:

$$\begin{aligned} Res(Supply(x_1, x_2, x_3), ic) &= \neg Class(x_3, x_4) \vee x_4 \neq t_1 \vee x_1 = s_1 \\ Res(Class(x_3, x_4), ic) &= \neg Supply(x_1, x_2, x_3) \vee x_4 \neq t_1 \vee x_1 = s_1 \end{aligned}$$

Therefore, the following rules are generated:

$$Supply(y_1, y_2, y_3) \mapsto Supply(y_1, y_2, y_3) \{ \forall y_4 [\neg Class(y_3, y_4) \vee y_4 \neq t_1 \vee y_1 = s_1] \}$$

$$Class(y_1, y_2) \mapsto Class(y_1, y_2) \{ \forall y_3, y_4 [\neg Supply(y_3, y_4, y_1) \vee y_2 \neq t_1 \vee y_3 = s_1] \}$$

The application of the operator T on the query $Q(z) = Class(z, t_1)$ gives:

- $T_0(Class(z, t_1)) = Class(z, t_1)$;
- $T_1(Class(z, t_1)) = Class(z, t_1) \wedge \forall x, y [T_1(\neg Supply(x, y, z)) \vee x = s_1]$;
- $T_2(Class(z, t_1)) = Class(z, t_1) \wedge \forall x, y [\neg Supply(x, y, z) \vee x = s_1]$.

At Step 2 the fixpoint is reached since the literal $Class(z, t_1)$ has been “expanded” and the literal $\neg Supply(x, y, z)$ does not have a residue associated to it ($T_1(\neg Supply(x, y, z)) = \neg Supply(x, y, z)$).

In order to answer the query $Q(z)$, the set of queries

$$T_\omega(Q(z)) = \{Class(z, t_1), \forall x, y [\neg Supply(x, y, z) \vee x = s_1]\}$$

is evaluated. Thus the answer is given by the query

$$Q'(z) = \text{Class}(z, t_1) \wedge \forall x, y [\neg \text{Supply}(x, y, z) \vee x = s_1]$$

which results in i_1 .

□

The operator $T_\omega(Q)$ conservatively extends the standard query evaluation on consistent databases. That is, given a set of integrity constraints IC and database D such that $D \models IC$, then for every query $Q(u)$ and tuple t , $D \models Q(t)$ if and only if $D \models T_\omega(Q(t))$.

Soundness and Completeness

In [4] the properties of *soundness*, *completeness* and *termination* of the operator T_ω have been investigated. Soundness means that every answer to $T_\omega(Q)$ is a consistent answer to Q . Completeness means that every consistent answer to Q is an answer to $T_\omega(Q)$. Termination means that there is an n such that for all $m \geq n$ $\forall X [T_n(Q(X)) \equiv T_m(Q(X))]$ is valid.

Sufficient conditions for soundness are that the query (having the form in (2.1)) is either (i) *universal* (i.e. it contains only universal quantifier) or (ii) *non-universal* and domain independent [74]. Thus for *non-universal* and domain dependent queries, like $Q = \exists x \neg P(x)$, soundness is not ensured.

Sufficient conditions for completeness are:

- i) queries that are conjunctions literals, i.e. quantifier-free sentence of the form

$$P_1(u_1) \wedge \dots \wedge P_i(u_i) \wedge \neg P_{i+1}(u_{i+1}) \wedge \dots \wedge \neg P_n(u_n) \wedge \varphi(u_1, \dots, u_n)$$

- ii) generic binary integrity constraints, i.e., sentence of the form

$$\forall X, Y [L_1(X) \vee L_2(Y) \vee \phi(X, Y)]$$

where L_1 and L_2 are literals and ϕ a formula containing only built-in predicates. *Generic* means that, it is required that the constraints does not imply any ground database literal, that is the constraints are not enough to answer a literal query by themselves. For instance, $\forall x [x = a \Rightarrow P(x)]$ is not *generic* since it forces $P(a)$ to hold.

In general, with disjunctive or existential queries completeness is not ensured. The following example shows a case in which the technique proposed is not complete.

Example 2.27 Consider the database D_1 and the integrity constraints of Example 2.21. The consistent answers to the (existential) boolean query $Q = \exists x_1, x_2 [\text{Student}(s_2, x_1, x_2)]$ is *true*, since Q is consistently true in every repair for D_1 .

But, it easy to see that $T_\omega(Q)$ is logically equivalent to

$$\exists x_1, x_2 [Student(s_2, x_1, x_2)] \wedge \forall x_3, x_4 [\neg Student(s_2, x_3, x_4) \vee x_1 = x_3] \wedge \\ \forall x_3, x_4 [\neg Student(s_2, x_3, x_4) \vee x_2 = x_4]$$

whose answer (in the original database instance) is *false*. Thus, the consistent answer *true* is not captured by the operator T_ω .

Similarly, the consistent answer to the disjunctive (ground boolean) query $Q = Student(s_2, John, Science) \vee Student(s_2, Frank, Engineering)$ is not captured by T_ω . □

Termination means that T_ω returns a finite set of formulas. It is important because then the set of consistent answers can be computed by evaluating a finite query. Termination can be guaranteed if there is an n such that $T_n(Q(X))$ and $T_{n+1}(Q(X))$ are syntactically the same. For any kind of query, it was shown that a necessary and sufficient condition ensuring this property is that the set constraints are *acyclic universal constraints* (cfr. Definition 1.2).

Therefore, the approach is sound, complete and finite for (quantifier-free) conjunction of literals queries and (general) binary acyclic universal constraints. In this case, the query transformation approach provide a direct way to establish PTIME-computability of consistent query answers. If the original query is first-order, so is the transformed version. In this way, we obtain a PTIME (or, more precisely AC^0) procedure for computing consistent query answers: transform the query and evaluate it in the original database. Note that the transformation of the query is done independently of the database instance and therefore, does not affect the data complexity.

2.6 Discussion

In this chapter we have presented some selected approaches which first investigated semantics of querying inconsistent databases obtained by integrating autonomous data.

The *flexible relational model* [2, 3] (cfr. Section 2.1) represents a way for augmenting the relational model by means of *ancillary attributes* for managing inconsistent (integrated) data w.r.t. primary key constraints. An extension of (classical) relations, namely *flexible relations*, has been introduced, but they are not in First Normal Form (1NF) (since contain sets of values for non-key attributes).

The limitation of dealing with only primary key dependencies and non-1NF relations has been subsequently overcome by the *integrated relational model* (cfr. Section 2.2) introduced in [35]. The notion of *possible integration* of relations [35] represents a first step towards the definition of repair of an inconsistent database, even if only consistency w.r.t. a set of key functional dependencies is considered and the extension of relations with null values is adopted (cfr. Section 2.2.2). Moreover, the modal operator \mathcal{K} in the *integrated*

relational calculus considers the intersection of answers posed on every possible consistent database belonging to the set of all *possible integration*. It represents a form of cautioning reasoning which is strictly correlated with the notion of *consistent query answer* [4] (cfr. Section 2.5).

Unlike the approaches above, the semantics of merging in [62] (cfr. Section 2.3) is defined for the general class of first-order integrity constraints. The semantics is based on the majority rule, i.e. an inconsistent tuple in the integrated database D will be in a repair for D if it is present in the majority of the sources to be integrated. When majority does not suffice to solve conflicts, the resulting database contains disjunctive information to be stored. Tables of OR-objects have been proposed for managing the resulting data, but no issues related to queries have been investigated.

In the approaches above, it is assumed that sources to be integrated are available. Specifically, in [2, 3] (cfr. Section 2.1) an ancillary attribute is used for identifying sources; in [35] (cfr. Section 2.2) the extension of relations by *null* values is performed starting from the original sources; in [62] (cfr. Section 2.3) the majority rule is based on the presence of the same tuple (of values) in a number of different sources. On the other hand, both the works [20] (cfr. Section 2.4) and [4] (cfr. Section 2.5) focus on an inconsistent database, possible obtained by integrating several sources which are not necessarily available. Indeed, the main issue addressed in these works is that of providing a method for distinguishing consistent and inconsistent answers.

In [20] (cfr. Section 2.4) a first notion of consistent answer has been proposed. The definition of consistent query answer is based on provability in minimal logic. The main intuition expressed is that the part of the database instance involved in an integrity violation should not be involved in the derivation of consistent query answers. But, in general, the data involved in an integrity violation is not entirely useless and reliable indefinite information can often be extracted from it. For instance, consider the database D such that $Facts(D) = \{P(a, b), P(a, c)\}$, and the functional dependency $FD : \forall x, y, z [P(x, y) \wedge P(x, z) \Rightarrow y = z]$. According to the definition of [20], for each query, there is no consistent answer in D , since $Kernel(D, FD) = D$. On the other hand, the answer to the boolean query $Q = \exists x P(a, x)$ should be *true*. This is captured by the definition of consistent query answer proposed in [4] (cfr. Section 2.5).

The model-theoretic notion of *consistent query answer* proposed in [4] parallels the standard notion of query answer in relational databases. It has been used and extended in [5, 7, 8, 9, 12, 16, 17, 23, 24, 27, 29, 41, 42, 44, 45, 46, 51, 52, 65, 66, 76, 77, 78]. An answer to query posed to a database that violates the integrity constraints is consistent if it is the same answer which is obtained if the query would be posed to any minimally repaired version of the original database. The semantics given for restoring consistency is that of minimal set of (whole) tuples which can be inserted or deleted in order to restore the consistency. Several alternative repair semantics have been proposed in literature: some of them are based on *only deletion* of whole tuples [27, 29,

44, 45, 46]; others focusing on *changing in attribute values* [16, 17, 39, 41, 42, 76, 77, 78].

In order to obtain consistent answers the *query rewriting* method has been proposed in [4], providing tractability of the consistent query answer problem for quantifier-free conjunction of literals queries and (generic) binary acyclic universal constraints. Given a query Q , the method computes a new query $T_\omega(Q)$ whose evaluation in a database returns the set of consistent answers to the original query Q . An implementation of the operator $T_\omega(Q)$ is presented in [24]. The technique of query rewritten was further extended in [45, 46] to a subset C_{tree} of conjunctive queries with existential quantification, under key constraints (cfr. Section 3.3 and 3.4).

The Tuple-Based Repairing Paradigm

The restoration of the consistency in a database should have a minimal impact on the original (inconsistent) database by trying to preserve as much information as possible. This can be accomplished in different ways, depending on whether the information in the database is assumed to be *correct* and *complete*. If the information is complete but not necessarily correct, i.e. it may violate a given set of integrity constraints, the only way to restore the database consistency is by *deleting* some part of it. If the information is both incorrect and incomplete, then both insertion and deletion should be considered.

As it will be clear in this chapter, for the class of *denial constraints* (which includes functional dependencies), the restriction to perform only deletions has no impact, since only deletions can remove integrity constraint violations. Indeed, in this case repairs are *maximal consistent subsets* of the original database instance, independently on the assumptions on the data. On the other hand, for inclusion dependencies repairs also consist of insertions of tuples, as we have observed in Section 2.5.1. Thus, in this case assumptions on data make the difference.

In this chapter we survey several works which present tractable and intractable cases for the problem of computing consistent answers in relational databases. These cases are characterized by the class of constraints, the class of queries and the repair-semantics employed. In [29] several classes of first-order queries and denial constraints for which both repair checking and consistent query answer problems are in *P*TIME have been provided. A rewriting algorithm for subclass of conjunctive queries (without repeated relation symbols) and key dependencies has been proposed in [45], and extended in [46] to work on Select-Projection-Join SQL queries. The complexity of the consistent query answer problem in presence of both functional and inclusion dependencies has been investigated under different repair-semantics in [29] and in [23].

Moreover, in this chapter the notion of *range-consistent query answer* [7, 8] for handling aggregation queries will be introduced. In [7, 8] the complexity of range-consistent query answers has been studied for aggregation queries without grouping. Then, aggregation queries with grouping have been subse-

quently investigated in [46], where a rewriting algorithm for computing range-consistent answers has been proposed.

In the approaches presented in this chapter, the task of repairing an inconsistent database is accomplished by working at tuple-level, i.e. inserting and/or deleting whole tuples. We call this repairing strategy *tuple-based repairing paradigm* and distinguish it from the *attribute-based* repairing paradigm (that will be introduced in Chapter 5), where a repair is obtained by modifying attribute values within tuples.

3.1 Range-Consistent Query Answers

In [7, 8] *scalar aggregation queries* in database that may violate a given set of functional dependencies have been studied. In this context, the definition of consistent query answers (which was first given for first-order queries) has been extended to such queries.

In defining consistent answers to aggregation queries it is distinguished between queries with *scalar* and *aggregation* functions. The former ones return a single value for the entire relation, whereas the latter ones perform grouping on a set of attributes and return a single value for each group. Both these kinds of query use the same standard set of SQL-2 aggregate operators: MIN, MAX, COUNT, SUM and AVG. In [7, 8] only *aggregation queries with scalar functions* on databases consisting of a single relation have been addressed.

Example 3.1 Consider the database D consisting of the relation *Salary*, whose instance is shown in Fig. 3.1, and the set of integrity constraints FD consisting of $Name \rightarrow Amount$.

<i>Name</i>	<i>Amount</i>
<i>Smith</i>	5000
<i>Smith</i>	8000
<i>Jones</i>	3000
<i>Stone</i>	7000

Fig. 3.1. Relation *Salary*

The relation *Salary* is inconsistent since there is a violation of the functional dependency, where participate the first two tuples. There are two repairs: the former is obtained deleting the first tuple, whereas the latter is obtained deleting the second tuple. If we pose the query `SELECT MIN(Amount) FROM Salary`, the value 3000 is given, independently of how the violation is fixed. That is, since the value 3000 come from a tuple that does not participate in any violation of the functional dependency, the evaluation of the function

$\text{MIN}(\textit{Amount})$ is the same in every repair, and 3000 is the consistent answer in the sense defined in [4].

Nevertheless, the query $\text{SELECT MAX}(\textit{Amount}) \text{ FROM } \textit{Salary}$ returns a different value in each repair: 8000 or 7000, respectively. Therefore, there is no consistent answer in the sense defined in [4]. \square

The example above suggests that for aggregation queries, the notion of consistent answer should be revisited to allow answers that are not single values, but intervals. In fact, under range-semantics for consistent query answers, the smallest possible range (the optimal lower and upper bounds) such that contains the answers given by all repairs has been defined as consistent answer.

Here we consider (scalar) aggregation queries of the form

$\text{SELECT } f \text{ FROM } P$

where f is one of: $\text{MIN}(A)$, $\text{MAX}(A)$, $\text{COUNT}(A)$, $\text{SUM}(A)$, $\text{AVG}(A)$ or COUNT^* , where A is an attribute of the relation scheme P .

These queries return a single numerical value by applying the corresponding *scalar function*, i.e. for $\text{MIN}(A)$, the minimum A -value in the given instance, etc. We also will denote with f an aggregation query (or a scalar function itself); thus, $f(I)$ will denote the result of applying f to the given instance I of P .

A *range-consistent answer to an aggregation query f* in a relation instance I w.r.t. a set of integrity constraints IC is the minimal range $\mathcal{R} = [a, b]$ such that for every repair R of P w.r.t. IC , the scalar value $f(R)$ of query f in R belongs to \mathcal{R} . The left and right end-points of the range \mathcal{R} are the *greatest lower bound* (glb) and *least upper bound* (lub), respectively, answers to f in R w.r.t. IC . They will be denoted as $\text{glb}_{IC}^f(R)$ and $\text{lub}_{IC}^f(R)$, respectively.

In the Example 3.1, the interval $[7000, 8000]$ is the consistent answer to the MAX query, whereas $[3000, 3000]$ is that of the MIN query.

The consistent query answer interval represents in a *succinct* form a superset of the values that the aggregation query can take in all possible repairs of the database w.r.t. a set of integrity constraints.

Now we define the data complexity of computing consistent answers under range-semantics which is useful for aggregation queries. Assume that a class of databases \mathcal{DC} , a class of aggregation queries \mathcal{AQ} and a class of integrity constraints \mathcal{IC} are given. The complexity of computing the *glb-answer* (resp. *lub-answer*) is defined to be the complexity of determining the membership of the sets

$$\mathcal{CQA}(IC, f) = \{(D, k) \mid D \in \mathcal{DC} \wedge \text{glb}_{IC}^f(D) \leq k\}$$

and

$$\mathcal{CQA}(IC, f) = \{(D, k) \mid D \in \mathcal{DC} \wedge \text{lub}_{IC}^f(D) \geq k\}$$

respectively, where IC is a fixed finite set of integrity constraints belonging to \mathcal{IC} , f is a fixed aggregation query belonging to \mathcal{AQ} and k a rational number.

For a complexity class \mathcal{C} , the problem $\mathcal{CQA}(IC, f)$ is \mathcal{C} -hard if there is a query $f_0 \in \mathcal{AQ}$ and a finite set of integrity constraints IC_0 such that $\mathcal{CQA}(IC_0, f_0)$ is \mathcal{C} -hard.

In the following, results are relative to a class \mathcal{AQ} containing only queries that use scalar functions of the same kind, e.g. $\text{MIN}(A)$ for some attribute A of a relation P . Moreover, we will deal with only unirelational databases and functional dependencies.

3.1.1 Conflict Graph

When only functional dependencies are considered, all repairs of an instance are obtained by deleting tuples from it, that is a repair is simply a maximal consistent subset of an instance. Clearly, there are only finitely many repairs since the relations are finite. Also, in this case the union of all repairs of an instance I is equal to I . These properties are not necessarily shared by classes of non-denial integrity constraints.

Given a set of functional dependencies FD and a database instance D , all repairs of D w.r.t. FD can be succinctly represented as a graph called *conflict graph*, denoted as $G_{FD,D}$. It is an undirect graph whose set of vertices is the set of tuples in D and whose set of edges consists of all edges (t_1, t_2) such that $t_1, t_2 \in D$ and there is a functional dependency $V \rightarrow W \in FD$ for which $t_1[V] = t_2[V]$ and $t_1[W] \neq t_2[W]$.

Example 3.2 Consider the relation *Salary* and the set of functional dependencies $FD = \{Name \rightarrow Amount\}$ of Example 3.1. The conflict graph $G_{FD,Salary}$ consists of four vertices (one for each tuple in *Salary*) and the edge $(\langle Smith, 5000 \rangle, \langle Smith, 8000 \rangle)$. □

An independent set S in an undirect graph $G = (V, E)$ is a subset of the set of vertices V of G , such that there is no edge in the set of edges E connecting two vertices in S . A *maximal independent set* is an independent set which is not a proper subset of any other independent set.

It is easy to verify that, given a set of functional dependencies FD and a database instance D , each repair of D w.r.t. FD corresponds to a maximal independent set in $G_{FD,D}$, and vice versa.

In addition to consistent answer, *glb-answer* and *lub-answer*, we can also consider *core answers*. A value v is a core answer to an aggregation query f in a relation P w.r.t. a set of functional dependencies FD if

$$v = f\left(\bigcap_{D' \in \mathcal{R}(D, FD)} D'\right)$$

where $\mathcal{R}(D, FD)$ is the set of repairs of D w.r.t. FD . The data complexity of computing core answers for any scalar function is in *PTIME* since the core

consists of all the isolated vertices in $G_{FD,D}$. Observe that, the notion of consistent answer in [20] (cfr. Section 2.4) corresponds to the notion of core answer.

The conflict graph can be exploited also for studying consistent answers and the *glb-answer* and *lub-answer* for aggregation queries.

3.1.2 Complexity of Scalar Aggregation Queries

In this section, first some results regarding the repair checking problem and the consistent query answer problem (cfr. Section 1.8) are presented. Then, complexity results for consistent answers for aggregation queries under range-semantics will be discussed.

In [8] it was shown that for a given set of functional dependencies FD , the complexity of checking whether a database instance D' is a repair for D is in *PTIME*. This means that the repair checking problem $\mathcal{RC}(FD)$ (cfr. Section 1.8) is tractable. It follows that for any set of FD and first-order query Q , the data complexity of checking whether a tuple t is a consistent answer to Q is in *coNP*. Actually, the consistent query answer problem $\mathcal{CQA}(Q, D, FD)$ is *coNP*-complete, since there is a set of two functional dependencies and a first-order query Q for which the problem of checking whether a tuple t is a consistent answer to Q is *coNP*-hard.

The Table 3.1 contains the complexity results for the problem of computing consistent answers to aggregation queries, i.e., the problem of checking whether the *glb-answer* to a query is $\leq k$ (respectively, the problem of checking whether the *lub-answer* to a query is $\geq k$).

	glb-answer		lub-answer	
	$ FD = 1$	$ FD \geq 2$	$ FD = 1$	$ FD \geq 2$
$\text{MIN}(A)$	<i>PTIME</i>	<i>PTIME</i>	<i>PTIME</i>	<i>NP</i> -complete
$\text{MAX}(A)$	<i>PTIME</i>	<i>NP</i> -complete	<i>PTIME</i>	<i>PTIME</i>
COUNT^*	<i>PTIME</i>	<i>NP</i> -complete	<i>PTIME</i>	<i>NP</i> -complete
$\text{COUNT}(A)$	<i>NP</i> -complete	<i>NP</i> -complete	<i>NP</i> -complete	<i>NP</i> -complete
$\text{SUM}(A)$	<i>PTIME</i>	<i>NP</i> -complete	<i>PTIME</i>	<i>NP</i> -complete
$\text{AVG}(A)$	<i>PTIME</i>	<i>NP</i> -complete	<i>PTIME</i>	<i>NP</i> -complete

Table 3.1. Complexity results for scalar aggregation queries

The *lub-answer* (resp. *glb-answer*) to a $\text{MAX}(A)$ (resp. $\text{MIN}(A)$) query in a relation P w.r.t. an arbitrary set of functional dependencies FD simply consists of evaluating the query in P , thus it is clearly in *PTIME*.

For all aggregate operators except $\text{COUNT}(A)$, the consistent query answer problem is in *PTIME* if the set of integrity constraints consists of at most one (nontrivial) functional dependency. For $\text{COUNT}(A)$, the consistent query answer problem is *NP*-complete, even for one functional dependency (it is

assumed that distinct values of some attribute A of a relation P are counted, i.e. it is equivalent to $\text{COUNT}(\text{DISTINCT}(A))$.

For a set of functional dependencies FD consisting of more than one functional dependency, the glb-problem (resp. the lub-problem) is NP -complete for all the aggregation queries (except that for lub-answer (resp. glb-answer) to $\text{MAX}(A)$ (resp. $\text{MIN}(A)$) queries, as we have already said above).

Computing glb and lub-answers with One Functional Dependency

For the aggregation queries $\text{MIN}(A)$, $\text{MAX}(A)$, $\text{COUNT}(*)$ and $\text{SUM}(A)$ and a single functional dependency the glb and the lub answers can be computed by SQL2 queries. This is in a sense analogue of the query transformation approach for first order queries [4] (cfr. Section 2.5). In fact, the resulting syntax of rewriting $\text{MIN}(A)$ and $\text{MAX}(A)$ queries as first-order queries does not allow the application of the methodology developed in [4] to them.

Consider the problem of computing the glb-answer in a relation P with a single functional dependency $V \rightarrow W$. For $\text{MAX}(A)$, the glb-answer can be defined in SQL2 as the following sequence of views:

```
CREATE VIEW S(V, W, C) AS
  SELECT V, W, MAX(A)
  FROM R
  GROUP BY V, W;
```

```
CREATE VIEW T(V, C) AS
  SELECT V, MIN(C)
  FROM S
  GROUP BY V;
```

```
SELECT MAX(C) FROM T
```

For $\text{SUM}(A)$, we have to replace MAX in the above by SUM . For $\text{COUNT}(*)$, we replace $\text{MAX}(A)$ by $\text{COUNT}(*)$ and $\text{MAX}(C)$ by $\text{SUM}(C)$.

Symmetric results hold for the lub-answer to a $\text{MIN}(A)$ query. Note that $\text{lub}_{FD}^{\text{MIN}(A)}(P) = -\text{glb}_{FD}^{\text{MAX}(A)}(P^-)$, where P^- is a relation containing identical tuples to P except that their A -values are inverted (every A -value v is changed to $-v$).

The lub-answer to $\text{MAX}(A)$, $\text{COUNT}(*)$ and $\text{SUM}(A)$ queries can be obtained in a similar way.

Example 3.3 Considering the relation *Salary* and the functional dependency $\text{Name} \rightarrow \text{Amount}$ of Example 3.1. The glb-answer to the query

```
SELECT MAX(Amount) FROM Salary
```

is computed by the following SQL query

```
SELECT MAX(A) FROM
  (SELECT MIN(Amount) AS A
   FROM Salary
   GROUP BY Name)
```

In this case we, can avoid to group for all attribute $(Name, Amount)$ of the relation since it has no effect, i.e. the view $S(V, W, C)$ is just the relation itself. □

For the aggregation query $AVG(A)$ the *PTIME* algorithm is iterative and can not be formulated in SQL2.

3.1.3 Other Tractable Cases

In [7, 8], some special properties of conflict graph which entail more tractable cases, for lub-answer to $COUNT(*)$ queries, were identified.

It has been shown that, given a relation scheme P is in Boyce-Codd Normal Form (BCNF), if the set of functional dependencies FD is equivalent to one with at most two dependencies, then computing lub-answer to $COUNT(*)$ queries in any instance of P can be done in *PTIME*.

Two approaches have been followed. The first is based on the observation that for a set FD of functional dependencies such that $|FD| = 2$ and a relation instance I whose scheme is in BCNF, the conflict graph $G_{FD,I}$ is *claw-free* and *perfect*. A graph is *claw-free* if it does not contain an induced subgraph (V_0, E_0) where $V_0 = \{t_1, t_2, t_3, t_4\}$ and $E_0 = \{(t_2, t_1), (t_3, t_1), (t_4, t_1)\}$. A graph is *perfect* if its chromatic number is equal to the size of its maximum clique. For such graphs computing a maximum independent set (an independent set of maximum cardinality) can be done in *PTIME*. Since repairs for I w.r.t. FD correspond to maximal independent sets in $G_{FD,I}$, the size of a maximum independent set provides the lub-answer to a $COUNT(*)$ aggregation query.

The second approach is based on computing maximum matching in a bipartite graph G' , such that matchings in G' one-to-one correspond to independent sets in the conflict graph $G_{FD,I}$. A time complexity bound of $\mathcal{O}(n^{1.5})$ has been provided, where n is the number of tuples in the relation.

If the set FD contains more than two dependencies the problem of computing lub-answer to $COUNT(*)$ queries become *NP-hard*, even if they are in BCNF.

There are other, simpler cases where the conflict graph has a structure that makes it possible to determine the cardinality of a maximum independent set in *PTIME*. For instance, if the relation instance I is the union of two instances that separately satisfy the set of functional dependencies FD , we obtain a bipartite conflict graph $G_{FD,I}$, for which determining a maximum independent

set can be done in *PTIME*. Observe that, a relation is obtained merging together two consistent relations in the context of database integration. Thus, also in this case the complexity of computing lub-answer to COUNT^* queries is in *PTIME*.

3.2 Repairing by Deleting a Minimal Set of Tuples

In [29] it is assumed that data are complete, but not necessarily correct. That is, a database instance may violate a given set of integrity constraints and the only way to restore the database consistency is by *deleting* (whole) tuples, since under the completeness assumption no new tuple should be inserted.

A repair is a database instance that satisfies the integrity constraints and minimally differs from the (possible inconsistent) original database. When we consider only deletions of complete tuples as ways to restore database consistency, the repairs are *maximal consistent subsets* of the original database instance.

Given a database scheme \mathcal{D} , a set of integrity constraints IC over \mathcal{D} and two database instance D and R over \mathcal{D} , we say that R is a repair for D w.r.t. IC under the repair-semantics of deletion of (whole) tuples if $Facts(R)$ is the *maximal subset* of $Facts(D)$ such that $Facts(R) \models IC$.

In [29] the computational complexity of the repair checking and the consistent query answers problem along several different dimensions has been studied, under repair-semantics of deletion of (whole) tuples. Specifically, the impact of the following parameters have been considered:

- the *class of queries*: quantifier-free queries, conjunctive queries, and simple conjunctive queries (conjunctive queries without repeated relation symbols and with limited variable sharing);
- the *class of integrity constraints*: denial constraints, functional dependencies, inclusion dependencies, and functional dependencies and inclusion dependencies together.
- the *number* of integrity constraints.

Several classes of queries and constraints for which both repair checking and consistent query answers are in *PTIME* has been obtained:

- i) quantifier-free queries and (arbitrary) denial constraints;
- ii) boolean simple conjunctive queries and functional dependencies (at most one per relation);
- iii) queries that are quantifier-free or simple conjunctive, and key functional dependencies and foreign key constraints, with at most one key per relation.

It was shown that repair checking, but not consistent query answers, are in *PTIME* for arbitrary functional dependencies and acyclic inclusion dependencies. Relaxing any of the above restrictions leads to *coNP*-hard problems.

(This, of course, does not preclude the possibility that introducing additional, orthogonal restrictions could lead to more *PTIME* cases). Moreover, it was shown that for arbitrary sets of functional dependencies and inclusion dependencies, repair checking is *coNP*-complete and consistent query answers is Π_2^P -complete.

3.2.1 Denial Constraints

The original notion of repair requires that the symmetric difference between a database and its repair be minimized. This is based on the assumption that the database may be not only inconsistent but also incomplete. The notion of repair pursued here reflects the assumption that the database is complete. For denial constraints integrity violations can only be removed by deleting tuples, so the different notions of repair in fact coincide in this case. Therefore, all the results presented in this Section are not affected by the restriction of the repairs to be subsets of the original instance. Insertions can restore integrity only for tuple-generating dependencies (e.g., inclusion dependencies).

Given a set of denial constraints DC and a database instance D , all the repairs of D with respect to DC can be succinctly represented as the *conflict hypergraph*. This is a generalization of the *conflict graph* defined in [7, 8] for functional dependencies only (see Section 3.1.1).

The conflict hypergraph $G_{DC,D}$ is a hypergraph whose set of vertices is the set $Facts(D)$ of facts of a database instance D and whose set of edges consists of all the sets $\{P_1(t_1), \dots, P_k(t_k)\}$ such that

- i) $P_1(t_1), \dots, P_k(t_k) \in Facts(D)$
- ii) there is a constraint $ic = \forall X_1, \dots, X_k \neg [P_1(X_1) \wedge \dots \wedge P_k(X_k) \wedge \varphi(X_1, \dots, X_k)]$ in DC such that $\{P_1(t_1), \dots, P_k(t_k)\}$ violate together ic , which means that there is a substitution θ such that $\forall i \in [1..k], \theta(X_i) = t_i$ and $\varphi(t_1, \dots, t_k)$ is true.

Note that there may be edges in $G_{DC,D}$ that contain only one vertex (it is the case of violations of *unary* denial constraints, where there is only one relation symbol). Moreover, the size of the conflict hypergraph is polynomial in the number of tuples in the database instance D .

By an *independent set* S in a hypergraph G we mean a subset of its set of vertices which does not contain any edge. An independent set S in a graph G is a *maximal independent set* if it is not a proper subset of any other independent set S' in G .

Each repair of a database instance D w.r.t. a set of denial constraints DC corresponds to a maximal independent set in $G_{DC,D}$. This entails the following two results, obtained as that in [7, 8] (cfr. Section 3.1.2) for functional dependencies only: for a given set of denial constraints DC and a boolean first-order query Q , (i) the data complexity of checking whether a database instance D' is a repair for D is in *PTIME*, (ii) the data complexity of deciding whether a tuple t is a consistent answer to Q is in *coNP*.

Moreover, a strategy for computing (nondeterministically) a repair for D w.r.t. DC is the following: pick a vertex of $G_{DC,D}$ which does not belong to a single-vertex edge and add vertices that do not result in the addition of an entire edge.

Tractable cases

By the *query rewriting* method, tractability of the consistent query answer problem for quantifier-free conjunction of literals queries and (generic) binary acyclic universal constraints has been provided in [4] (cfr. Section 2.5). Obviously, this result also holds when we restrict the class of constraints to binary denial constraints. In [29] it has been shown that for every set DC of denial constraints (not necessarily binary) and quantifier-free sentence Q (not necessarily a conjunction), the complexity of deciding whether a tuple t is a consistent answer to Q is still in *PTIME*.

The algorithm for *ground* quantifier-free sentence is shown in Fig. 3.2. It takes as input a quantifier-free formula Φ in Conjunctive Normal Form (CNF) and the conflict hypergraph $G_{DC,D}$ for a given database instance D and the set of denial constraints DC . It returns the consistent answer to Φ in D w.r.t. DC .

The first step of the algorithm reduces the task of determining whether *true* is the consistent query answer to the query Φ to answering the same question for every conjunct Φ_i . Then each formula Φ_i is negated and the rest of the algorithm attempts to find a repair R in which $\neg\Phi_i$ is *true* for some i . (we are checking if a ground clause Φ_i is *not* consistently *true* in D). Since each Φ_i is a disjunction of ground literals, $\neg\Phi_i$ is of the form $\neg P_{i_1}(t_1) \wedge \dots \wedge \neg P_{i_p}(t_p) \wedge P_{i_{p+1}}(t_{p+1}) \wedge \dots \wedge P_{i_m}(t_m)$.

The algorithm search for a repair such that:

- i) for each $j \in \{p+1, \dots, m\}$, t_j belongs to the instance of relation P_{i_j} ;
- ii) for each $j \in \{1, \dots, p\}$ a tuple t_j does not belong to the instance of relation P_{i_j} .

If exists $j \in \{p+1, \dots, m\}$ such that t_j does not belong to the relation instance P_{i_j} , then $\neg\Phi_i$ is *false* in every repair for D w.r.t. DC (Φ_i is *true*); thus the next conjunct Φ_{i+1} is examined.

Otherwise, all $P_{i_{p+1}}(t_{p+1}), \dots, P_{i_m}(t_m)$ are added to the set B of facts which represents the repair R we are constructing. Then, the algorithm (non deterministically) selects for every $j \in \{1, \dots, p\}$ such that t_j belongs to the instance of P_{i_j} , an hyperedge e_j of $G_{DC,D}$ containing the fact $P_{i_j}(t_j)$. This fact must be excluded from the repair R . If the set $B \cup (e_j - \{P_{i_j}(t_j)\})$ results in an independent set for $G_{DC,D}$, then B can be extended to a maximal one corresponding to a repair in which $\neg\Phi_i$ is *true* (and then Φ is *false*). If the algorithm does not succeed for any i (with $i \in \{1, \dots, k\}$), then *true* is the consistent query answer to Φ .

INPUT:
 $\Phi = \Phi_1 \wedge \dots \wedge \Phi_k$: quantifier-free sentence in CNF
 $G_{DC,D}$: conflict hypergraph of D w.r.t. DC

OUTPUT:
true if Φ is a consistently true in D , *false* otherwise

VAR:
 B : Set of facts
 $I(P)$: Instance of the relation P in D
 $E_{DC,D}$: Set of hyperedges in $G_{DC,D}$

begin

```

01)  for  $i \in \{1, \dots, k\}$  do
02)    let  $\neg\Phi_i \equiv \neg P_{i_1}(t_1) \wedge \dots \wedge \neg P_{i_p}(t_p) \wedge P_{i_{p+1}}(t_{p+1}) \wedge \dots \wedge P_{i_m}(t_m)$ .
03)    for  $j \in \{p+1, \dots, m\}$  do
04)      if  $t_j \notin I(P_{i_j})$  then
05)        next  $i$ ;
06)       $B \leftarrow \{P_{i_{p+1}}(t_{p+1}), \dots, P_{i_m}(t_m)\}$ 
07)      for  $j \in \{1, \dots, p\}$  do
08)        if  $t_j \in I(P_{i_j})$  then
09)          choose  $e_j \in \{e \in E_{DC,D} \mid P_{i_j}(t_j) \in e\}$ 
10)           $B \leftarrow B \cup (e_j - \{P_{i_j}(t_j)\})$ 
11)        if  $B$  is independent in  $G_{DC,D}$  then
12)          return false
13)  return true
end

```

Fig. 3.2. Algorithm for computing consistent answers to ground CNF queries

The algorithm in Fig. 3.2 need $m - p$ nondeterministic steps, a number which is independent of the size of the database (but dependent on Φ), 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 polynomial-time deterministic algorithm. Moreover, the assumption that Φ is a CNF formula does not affect the data complexity of the query evaluation, because every ground query can be converted to CNF independently of the (size of the) database. However, from a practical point of view, CNF conversion may lead to unacceptably complex queries.

The above approach can be generalized to any quantifier-free query, not necessarily ground. The idea is to design a *generator* of ground queries and use the above algorithm as a *checker*. In [27] it is shown how to find an appropriate set of binding for the variables in the asked (quantifier-free) formula. This is done by evaluating an *envelope* query over the database. An envelope query satisfies the following properties: (i) it returns a *superset* of the set of consistent query answers for every database instance, and (ii) it is easily constructible from the original query. Suppose that $E_Q(D)$ is the result

of evaluating an envelope query E_Q for a query Q in a database D . Then a tuple t is a consistent answers to Q in D (w.r.t. a set of constraints DC) if $q \in E_Q(D)$ and for every repair R of D (w.r.t. DC), $R \models Q(t)$.

In order to process a quantifier-free query Q , first, an envelope query E_Q is estimated, providing a superset of the consistent answers $E_Q(D)$. Then, for every tuple t in $E_Q(D)$, the grounding of Q is performed, obtaining a first-order ground query Q_t . Finally, the algorithm in Fig. 3.2 takes as input the query Q_t and checks if *true* is the consistent answer for this query, and depending of the result the tuple t is returned or not.

In [27, 28] a database middleware system, called Hippo, for computing consistent query answers based on this approach is described.

Now we consider a special case of denial constraints, namely functional dependencies. If the set of integrity constraints consists of only one functional dependency per relation the conflict hypergraph has a very simple form. It is a disjoint union of full multipartite graphs. If this single dependency is a key dependency then the conflict graph is a union of disjoint cliques. Although the conflict hypergraph has a very simple structure, the consistent query answers can be computed in polynomial time only for restricted classes of conjunctive queries.

In [29] it has been shown that for a set FD of k functional dependencies such that each dependency is defined over a different relation among P_1, \dots, P_k , computing consistent answers to *boolean simple conjunctive query* can be accomplished in *PTIME* (more precisely in AC^0 data complexity). Specifically, for each boolean simple conjunctive query Q , there exists a sentence Q' such that for every database instance D , a tuple t is a consistent answers to Q in D w.r.t. FD iff t is an answer to Q' in D . That is Q' is a rewriting of the query Q . Since Q' is a first-order query, which size is linear in the size of Q , it can be evaluated in polynomial time w.r.t. the size of D .

Also in the case of a *simple conjunctive query* Q , not necessarily boolean, such that it does not contain multiple occurrences of the same variable, the set of consistent answers to such a query can obtained by evaluating the transformed query Q' from which the appropriate quantifiers are dropped, thus still in polynomial time. This is because a formula $P(\dots, a \dots)$, where a is a constant, is equivalent to $\exists x P(\dots, x \dots) \wedge x = a$. The consistent answers to a non-ground query can be obtained by considering all possible bindings for the variables in the query and evaluating each ground query obtained in this manner. Those bindings are restricted to values coming from the appropriate columns in the database instance.

Intractable cases

In [29] it has been shown that for boolean conjunctive queries (i.e. relaxing the restriction that the query is simple) and for functional dependencies, the consistent query answer problem becomes *coNP*-complete. This result still

hold when key dependencies and conjunctive queries without repeated relation symbols (but with joins) are considered.

Moreover, there is a set of functional dependencies consisting of two functional dependencies and a boolean single-atom conjunctive query for which the consistent query answer problem is *coNP*-complete (single-atom means that there is one relation predicate symbol). This result was obtained first in a slightly weaker form (for non-key functional dependencies) in [7, 8] (cfr. Section 3.1.2). Finally, there exists a (single) denial constraint and a boolean single-atom conjunctive query for which the consistent query answer problem is *coNP*-complete.

The complexity results relative to the consistent query answer problem for classes of denial constraints and queries are summarized in Table 3.2.

	Quantifier-Free Queries	Boolean Single-Atom	Boolean Simple Queries	Boolean Queries
arbitrary <i>DCs</i>	<i>PTIME</i>	<i>coNP</i> -complete	<i>coNP</i> -complete	<i>coNP</i> -complete
1 <i>FD</i> per relation	<i>PTIME</i>	<i>PTIME</i>	<i>PTIME</i>	<i>coNP</i> -complete
2 <i>key FDs</i>	<i>PTIME</i>	<i>coNP</i> -complete	<i>coNP</i> -complete	<i>coNP</i> -complete

Table 3.2. Complexity of the consistent query answer problem for denial constraints

3.2.2 Inclusion Dependencies

In [29] it is assumed that the data in the database are complete (but possibly incorrect) as often occurs in data warehousing applications. Therefore repairs can be constructed using only tuple deletions. This restriction is also beneficial from the computational point of view, as we will see in this section.

Example 3.4 Consider a database with two relations $Employee(SSN, Name)$ and $Manager(SSN)$. Assume that the sets constraints defined are the following:

$$FD = \{SSN \rightarrow Name, Name \rightarrow SSN\}$$

$$ID = \{Manager[SSN] \subseteq Employee[SSN]\}$$

The instances for the two relations are shown in Fig. 3.3.

The database instance does not violate the inclusion dependency but violate both the functional dependencies. If we consider only the functional dependencies, there are two repairs: one obtained by removing the third tuple from $Employee$, and the other by removing the first two tuples from the same relation. However, the second repair violates the inclusion dependency. This can be fixed by removing the first tuple from $Manager$. So if we consider all the constraints, there are two *deletion-only* repairs R_1 and R_2 which are shown in Fig. 3.4 and Fig. 3.5, respectively.

<i>SSN</i>	<i>Name</i>
1234	<i>Smith</i>
5555	<i>Jones</i>
5555	<i>Smith</i>

<i>SSN</i>
1234
5555

Employee *Manager*

Fig. 3.3. Relations *Employee* and *Manager*

<i>SSN</i>	<i>Name</i>
1234	<i>Smith</i>
5555	<i>Jones</i>

<i>SSN</i>
1234
5555

Employee *Manager*

Fig. 3.4. Repair R_1

<i>SSN</i>	<i>Name</i>
5555	<i>Smith</i>

<i>SSN</i>
5555

Employee *Manager*

Fig. 3.5. Repair R_2

If we consider repairs where both tuple insertion and deletions are permitted, then insertions may lead to infinitely many repairs of the form in Fig. 3.6, where c is an arbitrary string different from *Smith* (this is forced by one of the functional dependencies).

<i>SSN</i>	<i>Name</i>
1234	c
5555	<i>Smith</i>

<i>SSN</i>
1234
5555

Employee *Manager*

Fig. 3.6. Repair considering also tuple insertions

□

For a database instance D and a set of inclusion dependencies ID there is a single repair for D w.r.t ID , which is obtained by deleting all the tuples violating ID (and only those). Then for every set ID and boolean query Q ,

the computational complexity of both repair checking and consistent query answer problem is in *PTIME*.

We consider now functional and inclusion dependencies together. We identify the cases where both repair checking and computing consistent query answers can be done in *PTIME*. The intuition is to limit the interaction between the two type of dependencies in the given set of integrity constraints in such a way that the polynomial-time results presented for functional dependencies only can be exploited.

Let $IC = KD \cup FK$ be a set of constraints consisting of a set of key dependencies KD and a set of foreign key constraints FK , but with no more than one key per relation. Let D be a database instance and D' be the unique repair of D w.r.t. FK . Then

1. R is a repair of D w.r.t. IC if and only if R is a repair of D' w.r.t. KD ;
2. the repair checking problem is in *PTIME*;
3. for boolean quantifier-free queries or boolean simple conjunctive queries, the consistent query answer problem is in *PTIME*.

Observe that, repairing D' with respect to key constraints does not lead to new inclusion violations. This is because the set of key values in each relation remains unchanged after such a repair (which is not necessarily the case if we have relations with more than one key). The repairs for D w.r.t. IC are computed by repairing D w.r.t. FK and then repairing the result w.r.t. KD (which can be done in *PTIME* for all denial constraints). Under the assumption above, the polynomial-time results about consistent query answers obtained for functional dependencies only (see Table 3.2) can be transferred to the case examined.

For the cases that follows, the consistent query answers becomes a *coNP*-hard problem. Let $IC = FD \cup AID$ be a set of constraints consisting of a set of functional dependencies FD and an *acyclic* set of inclusion dependencies AID (cfr. Definition 1.3). In these cases, although the repair checking problem is in *PTIME*, the consistent query answer problem is *coNP*-hard for ground atomic queries (even if FD consists of only *key* functional dependencies and AID consist of *acyclic primary foreign key* constraints).

Relaxing the acyclicity assumption, even though only *one functional dependency* and *one inclusion dependency* are considered, the intractability of the repair checking problem (thus also for consistent query answer) remains. Moreover, considering *key* functional dependencies and *foreign key* constraints together, the repair checking problem is yet *coNP*-hard.

3.3 Rewriting for a Class of Conjunctive Queries

In [45] the problem of retrieving consistent answers over databases that might be inconsistent with respect to primary key constraints has been addressed. An algorithm that produces a first-order query rewriting for the problem of

computing consistent answers has been proposed. The algorithm works for a class of conjunctive queries, running in polynomial time in the size of the query. Specifically, this class of query is defined in terms of the *join graph* of the query. The join graph is a directed graph such that: its vertices are the literals of the query; and it has an edge for each join in the query that involves some variable at the position of a non-key attribute. The algorithm works for conjunctive queries without repeated relation symbols (but with any number of literals and variables) whose join graph is a forest.

Moreover, in [45] it has been shown a class of conjunctive queries such that the problem of computing the consistent answers is *coNP*-complete for *every* query of the class whose join graph is not a forest. This type of result is much stronger than the usual approach taken in literature, which consists of showing intractability of a class by exhibiting at least one query for which the problem is intractable. A dichotomy for this class of queries has been provided: given a query Q , either computing the consistent answers for Q is in *PTIME* (it is first-order rewritable) or it is a *coNP*-complete problem.

3.3.1 The Class of Tree Queries

In [45] it has been assumed that the set of integrity constraints consists of at most one key dependency per relation. Moreover, we focus on conjunctive queries without repeated relation symbols having the form

$$Q(w_1, \dots, w_m) = \exists z_1, \dots, z_k [P_1(u_1) \wedge \dots \wedge P_n(u_n)] \quad (3.1)$$

where P_1, \dots, P_n are distinct relation symbols.

The problem of computing consistent answers for conjunctive queries over databases that might violate a set of key constraints is *coNP*-complete for conjunctive queries, even though queries have no repeated relation symbol [29] (cfr. Section 3.2.1). However, this does not necessarily preclude the existence of classes of queries for which the problem is easier to compute. In fact, in [45] the class \mathcal{C}_{tree} of conjunctive queries for which the problem of computing consistent answers is tractable has been introduced. In order to define the syntactic conditions that the queries in such class must satisfy, we introduce the *join graph* of the query.

Let Q be a conjunctive query of the form (3.1). The join graph G_Q of Q is a directed graph such that:

- i) the vertices of G_Q are the literals of Q ;
- ii) there is an edge from P_i to P_j (i.e there is $\langle P_i, P_j \rangle$) if $i \neq j$ and there is some variable x such that x occurs at the position of a non-key attribute in $P_i(u_i)$ and x occurs (anywhere) in $P_j(u_j)$;
- iii) there is a self-loop at P_i (i.e., an edge from P_i to P_i) if there is some variable x such that x occurs at the position of a non-key attribute of P_i , and x occurs at least twice in $P_i(u_i)$.

Example 3.5 Consider the relation schemes $P_i(A_i, B_i)$ where $pkey(P_i) = \{A_i\}$, with $1 \leq i \leq 4$ (the only constraints defined are primary key dependencies $A_i \rightarrow B_i$). Let Q_1 be the query

$$Q_1 = \exists x_1, x_2, x_3, x_4 [P_1(x_1, x_2) \wedge P_2(x_2, x_3) \wedge P_3(x_3, x_4) \wedge P_4(x_2, a)]$$

The join graph of Q_1 consists of a tree whose root is P_1 : there are the edges (P_1, P_2) (y occurs at the position of a non-key attribute in P_1 and y occurs in P_2), (P_2, P_3) and (P_1, P_4) . □

We will focus on queries whose join graph is a forest (a set of tree). Moreover, we will impose the additional condition that the joins from non-key to key attributes involve the *entire* key of a relation. We will call such joins *full*. Let $P_i(X_i, Y_i)$ and $P_j(X_j, Y_j)$ be a pair of literals of a conjunctive query Q . Assume that X_i (resp. X_j) are all the variables at the positions of the key attribute of P_i (resp. P_j). We say that there is a *full non-key to key join* from P_i to P_j if every variable of X_j appears in Y_i .

Example 3.6 All the non-key to key joins of the query Q_1 in the Example 3.5 are *full* non-key to key joins. On the other hand, consider the relation scheme $P_1(A, B)$ and $P_2(C, D, E)$ with $pkey(P_1) = \{A\}$ and $pkey(P_2) = \{C, D\}$. Given the query

$$Q_2 = \exists x_1, x_2, x_3, x_4 [P_1(x_1, x_2) \wedge P_2(x_2, x_3, x_4)]$$

the join on the variable x_2 between P_1 and P_2 is not full since it does not involve all the variables at the position of the key of P_2 . □

The class \mathcal{C}_{tree} of conjunctive queries for which the problem of computing consistent answers is tractable is defined as follows. Let Q be conjunctive query without repeated relation symbols. Let G_Q be the join graph of Q . We say that $Q \in \mathcal{C}_{tree}$ if G_Q is a forest (i.e., every connected component of G_Q is a tree) and every non-key to key join of Q is full.

Example 3.7 The query Q_1 of the Example 3.5 is in \mathcal{C}_{tree} , whereas the query Q_2 in the Example 3.6 is not in \mathcal{C}_{tree} , since it has a non-key to key join which is not full.

Consider the relation schemes $P_1(A, B)$ and $P_2(C, D, E)$ with $pkey(P_1) = \{A\}$ and $pkey(P_2) = \{C, D\}$. The query

$$Q_3 = \exists x_1, x_2, x_3 [P_1(x_1, x_2) \wedge P_2(x_2, x_3, x_1)]$$

is not in \mathcal{C}_{tree} since its join graph contains a cycle: there is an edge (P_1, P_2) (x_2 occurs at the position of a non-key attribute in P_1 and x_2 occurs in P_2) and an edge (P_2, P_1) (x_1 occurs at the position of a non-key attribute in P_2 and x_1 occurs in P_1). Moreover, the join on the variable x_2 is not full since it does not involve the entire key of P_2 . □

3.3.2 The Query Rewriting Algorithm

The following two examples highlight the intuition underlying the query rewriting algorithm.

Example 3.8 Consider the relation scheme $P(A, B)$ where $pkey(P) = A$. Assume that the relation instance I_1 is $\{P(a_1, b_1), P(a_1, b_2)\}$. Given the query $Q = \exists x P(x, b_1)$, the consistent answer to Q is *false* in I_1 . Now, consider the instance $I_2 = \{P(a_1, b_1), P(a_1, b_2), P(a_2, b_1)\}$. It is easy to see that the consistent answer to Q is *true* in I_2 . This is because there is a key value in P (a_2 in this case) that appears with b_1 as its non-key value, and does not appear with any other constant b such that $b \neq b_1$. The latter condition can be checked by the formula $Q_{consist}(x) = \forall y P(x, y) \Rightarrow y = b_1$. Therefore, the query rewriting algorithm yields the following query:

$$Q' = \exists x P(x, b_1) \wedge \forall y P(x, y) \Rightarrow y = b_1$$

□

Example 3.9 Consider the relation schemes $P_1(A_1, B_1)$ and $P_2(A_2, B_2)$ where $pkey(P_1) = A_1$ and $pkey(P_2) = A_2$. Assume that the instance I_1 is $\{P_1(a_1, b_1), P_1(a_1, b_2), P_2(b_1, c_1)\}$. It is easy to verify that the consistent answer to the query $Q = \exists x_1, x_2, x_3 P_1(x_1, x_2) \wedge P_2(x_2, x_3)$ is *false*. On the other hand, if we consider the instance I_2 containing the facts $\{P_1(a_1, b_1), P_1(a_1, b_2), P_2(b_1, c_1), P_2(b_2, c_2)\}$, the consistent answer to Q is *true*. This occurs because every non-key value that appears together with a_1 in some tuple (in this case, b_1 and b_2) joins with a tuple of P_2 . The formula

$$Q_{consist}(x'_1) = \forall x'_2 [P_1(x_1, x'_2) \Rightarrow \exists x'_3 P_2(x'_2, x'_3)]$$

checks the condition above. The query rewriting algorithm yields the query

$$Q' = \exists x_1, x_2, x_3 P_1(x_1, x_2) \wedge P_2(x_2, x_3) \wedge \forall x'_2 [P_1(x_1, x'_2) \Rightarrow \exists x'_3 P_2(x'_2, x'_3)]$$

□

Given a query Q such that $Q \in \mathcal{C}_{tree}$, and a set of primary key functional dependencies KD , the rewriting algorithm returns a first-order rewritten query Q' for the problem of obtaining the consistent answers for Q w.r.t. KD . The query Q' is obtained as the conjunction of the input query Q , and a new query called $Q_{consist}$, which is used to ensure that Q is satisfied in every repair for the database instance w.r.t. KD (thus the consistent answer is *true*). The query $Q_{consist}$ is obtained by recursion on the tree structure of each of the components of the join graph G_Q of Q , which is a forest.

The algorithm is applied to queries with free variables, as $Q(y) = \exists x P(x, y)$, in a similar way to the case of queries with constants, as that in the Example 3.8. It suffices to treat the free variable as if they were constants. For instance, for $Q(y)$ the algorithm yields the rewritten query

$$Q'(y) = \exists x P(x, y) \wedge \forall y' P(x, y') \Rightarrow y' = y$$

where the only difference with the query in the Example 3.8 is that the constant a is replaced by the free variable y .

3.3.3 A Dichotomy Result

The rewriting algorithm works for a set integrity constraints consisting of exactly a key per relation and queries with full joins whose join graph is a forest. This is a sufficient condition for a query to be first-order rewritable. In [45] it has been shown a class \mathcal{C}^* of queries such that the problem of computing the consistent answers is *coNP*-complete for *every* query of the \mathcal{C}^* which does not satisfy the conditions of the rewriting algorithm (i.e. the membership in \mathcal{C}_{tree}).

We say that a conjunctive query Q without repeated relation symbols is in the class \mathcal{C}^* if the following conditions hold:

- i) for every literal $P(X, Y)$ of Q where X is the set of variables at the positions of the key of P , there is x in X such that it does not belong to the set X' of variables at the positions of the key of any literal P' of Q , with $P' \neq P$;
- ii) the join graph G_Q of Q has no self-loops;
- iii) if the join graph G_Q of Q is a forest, then every non-key to key join of Q is full.

Observe that, the first condition excludes queries having different literals with the same key, like $\exists x_1, x_2, x_3 P_1(x, y) \wedge P_2(x, y)$ where x is the variable at the position of the key attribute of both P_1 and P_2 . The second condition excludes queries like $\exists x P(x, x)$, where x is the variable at the position of the key of P . Finally, the case of queries whose join graph is a forest but involving *not-full* (i.e. partial) non-key to key joins has been left out of \mathcal{C}^* . For the conjunctive queries which does not belong to \mathcal{C}^* there has not been provided the dichotomy result.

Let \mathcal{C}_{hard} the class of the query Q such that $Q \in \mathcal{C}^*$ and $Q \notin \mathcal{C}_{tree}$. It has been shown that the problem of computing consistent query answer to $Q \in \mathcal{C}_{hard}$ w.r.t. a set of key dependencies is *coNP*-complete. Therefore, given a query Q such that $Q \in \mathcal{C}^*$, it can be decided in polynomial time whether the problem of computing consistent answer to Q is either in *PTIME* or it is *coNP*-complete.

Example 3.10 Consider the relations $P_i(A_i, B_i)$ and primary key dependencies $A_i \rightarrow B_i$ with $1 \leq i \leq 3$. Then the query

$$Q_4 = \exists x_1, x_2, x_3, x_4 [P_1(x_1, x_2) \wedge P_2(x_3, x_4) \wedge P_3(x_2, x_4)]$$

is in \mathcal{C}^* , but it is not in \mathcal{C}_{tree} since its join graph is not a tree (the vertex P_3 has two incoming edges: the first from P_1 and the second from P_2). Therefore, $Q_4 \in \mathcal{C}_{hard}$.

As second example, also the query Q_3 of the Example 3.7 belongs to the class \mathcal{C}_{hard} .

□

3.4 Rewriting SQL Queries

In [46] the rewriting algorithm presented in [45] has been improved and extended for working on Select-Projection-Join (SPJ) SQL queries (with set semantics) and also SPJ queries with aggregation, grouping, and bag semantics.

The algorithm introduced in [45] (cfr. Section 3.3.2) aim to produce a first-order rewriting of conjunctive queries for the problem of computing consistent answer in databases that may violate a set of key functional dependencies. In principle, first-order queries can be translated into SQL queries. However, the queries produced by means of the algorithm in [45] have a high level of nesting (proportional to the number of relations in the query) and are therefore very inefficient. The rewriting algorithm proposed in [46] produces SQL queries with at most one level of nesting, which has reasonable running times.

Moreover, an algorithm working for aggregation queries with grouping has been proposed. The first works on aggregation queries were that of [7, 8], where it has been proposed the use of ranges as a semantics for consistent answering for aggregate queries 3.1. In [7, 8] only queries with just one aggregated attribute and no grouping were considered, namely *scalar* aggregation queries. On the other hand, in [46] these results was extended to consider aggregation queries with grouping, even if it is yet limited to consider only one relation in the FROM clause as in [7, 8].

A system, called ConQuer (it stands for Consistent Querying) [46, 47], for managing inconsistent data has been proposed. In ConQuer, a user may postulate a set of key functional dependencies, possibly at query time, and the system retrieves exactly the query answers that are consistent w.r.t. the key constraints. In order to do this, ConQuer rewrites the query into another SQL query that retrieves the consistent answers.

The class of SQL queries, called *tree queries*, that can be handled by ConQuer is an extension of the class \mathcal{C}_{tree} of conjunctive queries introduced in Section 3.3.2. The extensions consists in the presence of comparison operators in the selection condition and aggregate expressions.

In order to define a *tree query*, we consider the *join graph* for an SQL query, similar to that introduced in the Section 3.3.1 for conjunctive queries.

The join graph G_Q of an SQL query Q is a directed graph such that:

- the vertices of G_Q are the relations used in the FROM clause of Q ;
- there is an edge from P_i to P_j if a non-key attribute of P_i is equated with a key attribute of P_j (the key attribute of P are that belonging to $pkey(P)$).

Differently from the join graph of Section 3.3.1, here we do not consider self-loops because of only joins which involve at least an entire key of one relation are examined (these correspond to *full* non-key to key joins).

A select-project-join-group-by query Q is a *tree query* if

- i) there are no repeated relation symbols in the **FROM** clause of Q ;
- ii) every join condition of Q is an equi-join (i.e. there are no inequality joins);
- iii) every join involves the key of at least one relation, (i.e. non-key to non-key joins are not permitted);
- ii) the join graph G_Q of Q is a tree.

Observe that the **WHERE** clause of Q may contain conjunction of comparison predicated (e.g. $<$, \leq). However, a tree query does not contain neither nested subqueries nor disjunctions.

3.4.1 Join Queries

We now illustrate the rewriting technique for tree queries without aggregation or grouping by means of an example.

Example 3.11 Consider the database consisting of the two relation schemes $Customer(Cust, Account)$ and $Order(Cod, Clerk, Cust)$, whose instances are shown in Fig. 3.7. Assume that $pkey(Customer) = \{Cust\}$ and $pkey(Order) = \{Cod\}$.

<i>Cust</i>	<i>Account</i>
c_1	2000
c_1	100
c_2	2500
c_3	2200
c_3	2500

Customer

<i>Cod</i>	<i>Clerk</i>	<i>Cust</i>
o_1	<i>ali</i>	c_1
o_2	<i>jo</i>	c_2
o_2	<i>ali</i>	c_3
o_3	<i>ali</i>	c_4
o_3	<i>pat</i>	c_2
o_4	<i>ali</i>	c_2
o_4	<i>ali</i>	c_3
o_5	<i>ali</i>	c_2

Order

Fig. 3.7. Relation instances *Customer* and *Order*

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 $C.Account > 1000$ AND $O.Cust = C.Cust$;

The consistent query answers for Q is ali which has (certainly) processed the orders o_4 and o_5 . □

The query rewriting of Q is obtained using two subqueries, namely *Candidates* and *Filter*, as follows:

```
SELECT Clerk
FROM Candidates Cand
WHERE NOT EXISTS (SELECT *
                  FROM Filter F
                  WHERE Cand.Cust = F.Cust)
```

The (sub)query *Candidates* returns all distinct pair $\langle Cod, Clerk \rangle$ that satisfies the selection conditions of the original query Q , and thus possibly belongs to the consistent answers. Therefore, it corresponds to the original query, except that it selects distinct pairs $\langle Cod, Clerk \rangle$ (in general, the argument of the **SELECT** clause of *Candidates* is augmented with the key attributes of the root of the join graph G_Q , i.e. the key of *Order* in this case). It is defined as follows:

```
Candidates AS ( SELECT DISTINCT O.Cod, O.Clerk
                FROM Customer C, Order O
                WHERE C.Account > 1000 AND O.Cust = C.Cust)
```

In this case, the result of applying the subquery *Candidates* to the databases is $\{\langle o_1, ali \rangle, \langle o_2, jo \rangle, \langle o_2, ali \rangle, \langle o_3, pat \rangle, \langle o_4, ali \rangle, \langle o_5, ali \rangle\}$.

The (sub)query *Filter* returns the orders that should be “filtered out” from the result of *Candidates* because they are not consistent answers. It is defined as follows:

```
Filter AS ( SELECT O.Cod
            FROM Candidates Cand
            JOIN Order O ON Cand.Cod = O.Cod
            LEFT OUTER JOIN Customer C ON O.Cust = C.Cust
            WHERE C.Cust IS NULL OR C.Account ≤ 1000)
UNION ALL
SELECT Cod
FROM Candidates Cand
GROUP BY Cod
HAVING COUNT(*) > 1)
```

In this case, *Filter* returns the orders $\{o_1, o_2, o_3\}$. The orders o_1 and o_3 are filtered out by the former subquery of *Filter*, whereas the order o_2 by the latter.

- The order o_1 is returned by *Filter* because the first tuple of *Order* joins with the second tuple of *Customer*, which corresponds to a customer whose account balance is below 1000.
- The order o_3 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 o_3 appears together with a *null* value for attribute *Cust* in the left-outer join.
- The order o_2 is returned by *Filter* because the clerk of o_2 may be *jo* in some repairs, and *ali* in others. Hence, o_2 should not contribute with its clerks to the consistent query answer of Q .

In [46], the rewriting above has been generalized for obtaining consistent answers for tree queries without aggregate operators w.r.t. a set of integrity constraints containing at most one key functional dependencies per relation.

Observe that, detecting of the cases in which non-key to key joins are not satisfied (as for order o_3) 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.

3.4.2 Aggregation Queries

The aggregation queries considered in [46] are SQL queries of the form:

```
SELECT  $G$ ,  $aggr_1(e_1)$  AS  $E_1$ , ...,  $aggr_n(e_n)$  AS  $E_n$ ,
FROM  $P$ 
WHERE  $\mathcal{W}$ 
GROUP BY  $G$ 
```

where G is the set of attributes we are grouping on, and $aggr_1(e_1), \dots, aggr_n(e_n)$ are aggregate expressions with functions $aggr_1, \dots, aggr_n$, respectively. Each $aggr_i$ ($1 \leq i \leq n$) is in $\{\text{MIN}, \text{MAX}, \text{SUM}\}$. Note that, 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 adopted for consistent query answers to queries with aggregate expressions is the range-semantics (cfr. Section 3.1) that has been proposed in [7]. In that work queries with just one aggregated attribute and no grouping were considered (cfr. Section 3.1.2). Thus a single range has been

defined, which contains every values that the (scalar) aggregation query can take in all possible repairs of the database. Here, queries with n aggregated attribute and grouping are considered, therefore n ranges for each value of G that is a consistent answer will be used.

Consider the query Q_G that consists of Q with all the aggregate expression removed from the **SELECT** clause, i.e. Q_G is of the form:

```
SELECT G
FROM P
WHERE W
GROUP BY G
```

A *range-consistent query answer* for an aggregation query Q on a database D w.r.t. a set of integrity constraints IC is a pair $\langle t, r \rangle$ such that

1. t is a consistent answer for Q_G in D ;
2. $r = \langle glb_{E_1}, lub_{E_1}, \dots, glb_{E_n}, lub_{E_n} \rangle$, and for each $i \in \{1, \dots, n\}$ the following conditions hold:
 - i) for every repair R of D w.r.t IC , $glb_{E_i} \leq \pi_{E_i}(\sigma_{G=t}(Q(R))) \leq lub_{E_i}$, where $\pi_{E_i}(\sigma_{G=t}(Q(R)))$ is the evaluation of the aggregate expression E_i on the group identified by t which is returned by the evaluating Q in R ; glb_{E_i} and lub_{E_i} are a lower bound and an upper bound, respectively;
 - ii) for some repair R , $\pi_{E_i}(\sigma_{G=t}(Q(R))) = glb_{E_i}$, i.e. glb_{E_i} is the greater lower bound;
 - iii) for some repair R , $\pi_{E_i}(\sigma_{G=t}(Q(R))) = lub_{E_i}$, i.e. lub_{E_i} is the least upper bound;

The definition of range-consistent query answer in [46] extends the definition of [7, 8] to queries with more than an aggregation expression, and with the **WHERE** and the **GROUP BY** clause (both the works refer to **FROM** clauses with one relation).

Example 3.12 Consider the database consisting of the relation $Customer(Cust, Nation, Segment, Account)$ whose instance is shown in Fig. 3.8. Assume that key of $Customer$ is the attribute $Cust$.

<i>Cust</i>	<i>Nation</i>	<i>Segment</i>	<i>Account</i>
c_1	n_1	<i>building</i>	1000
c_1	n_1	<i>building</i>	2000
c_2	n_1	<i>building</i>	500
c_2	n_1	<i>banking</i>	600
c_3	n_2	<i>banking</i>	100

Fig. 3.8. Relation $Customer$

Consider the following query Q , which retrieves the total account balance for customers in the building sector, grouped by nation.

```
SELECT Nation, SUM(Account)
FROM Customer
WHERE Segment = 'building'
GROUP BY Nation
```

Then the query Q_G is as follows.

```
SELECT Nation
FROM Customer
WHERE Segment = 'building'
GROUP BY Nation
```

There are four repairs for $Customer$ w.r.t. the key constraint:

$$R_1 = \{ \langle c_1, n_1, building, 1000 \rangle, \langle c_2, n_1, building, 500 \rangle, \langle c_3, n_2, banking, 100 \rangle \}$$

$$R_2 = \{ \langle c_1, n_1, building, 1000 \rangle, \langle c_2, n_1, banking, 600 \rangle, \langle c_3, n_2, banking, 100 \rangle \}$$

$$R_3 = \{ \langle c_1, n_1, building, 2000 \rangle, \langle c_2, n_1, building, 500 \rangle, \langle c_3, n_2, banking, 100 \rangle \}$$

$$R_4 = \{ \langle c_1, n_1, building, 2000 \rangle, \langle c_2, n_1, banking, 600 \rangle, \langle c_3, n_2, banking, 100 \rangle \}$$

The consistent answer to the query Q_G is $\{n_1\}$. The result of applying Q to these repairs is the following: $Q(R_1) = \{ \langle n_1, 1500 \rangle \}$, $Q(R_2) = \{ \langle n_1, 1000 \rangle \}$, $Q(R_3) = \{ \langle n_1, 2500 \rangle \}$, $Q(R_4) = \{ \langle n_1, 2000 \rangle \}$. Therefore the *range-consistent query answers* is the set $\{ \langle n_1, \langle 1000, 2500 \rangle \rangle \}$. \square

We now illustrate the rewriting technique for queries with aggregation and grouping. We illustrate the approach for computing range-consistent answer by means of the database instance ($Customer$) and the aggregation query Q of the example above.

The tuple of the relation $Customer$ which produce the consistent answers for Q_G are the first and the second tuple, which contribute with the nation n_1 .

In order to obtain the range-consistent answer to Q , we separately consider the set of tuples which contributes to the consistent answers for Q_G and the set which does not contribute. This distinction is achieved by means of the subquery $Filter$ for the join query Q_G , which can be obtained as seen in Section 3.4.1. Intuitively, the filter retrieves the customers which appear in some tuple that does not satisfy Q_G . In this case, the filtered out customers are c_2 and c_3 . The customer c_1 is not filtered because its two tuples (the first and the second in $Customer$) satisfy query Q_G .

In the repairs R_1 and R_2 , c_1 contributes an account balance of 1000. Whereas in R_3 and R_4 , it contributes 2000. Therefore, it contributes a minimum of 1000 and a maximum of 2000. This is captured by means of the following query:

```

UnFilteredCandidates AS (
  SELECT Cust, Nation,
         MIN(Account) AS MinAcc, MAX(Account) AS MaxAcc
  FROM Customer C
  WHERE Segment = 'building'
         AND NOT EXISTS ( SELECT *
                          FROM Filter
                          WHERE C.Cust = Filter.Cust)
  GROUP BY Cust, Nation)

```

The result of applying *UnFilteredCandidates* to the inconsistent database is $\{c_1, n_1, 1000, 2000\}$.

In order to compute the contribute of the tuples that are filtered out (c_2 and c_3), we consider the following subquery:

```

FilteredCandidates AS (
  SELECT Cust, Nation, 0 AS MinAcc, MAX(Account) AS MaxAcc
  FROM Customer C
  WHERE Segment = 'building'
         AND EXISTS ( SELECT *
                      FROM Filter
                      WHERE C.Cust = Filter.Cust)
         AND EXISTS ( SELECT *
                      FROM QGCons
                      WHERE C.Nation = QGCons.Nation)
  GROUP BY Cust, Nation)

```

The result of *FilteredCandidates* is $\{c_2, n_1, 0, 500\}$. The customer c_3 is not returned since its nation (i.e., the attribute in the group by of the original query) does not appear in the result of the consistent answer 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.

The range-consistency answers are obtained summing the contributes (lower and upper bounds) of each nation in the result of *UnFilteredCandidates* and *FilteredCandidates*, as follows:

```

SELECT Nation, SUM(MinAcc) AS glb, SUM(MaxAcc) AS lub
FROM ( SELECT *
       FROM UnFilteredCandidates
       UNION

```

```

SELECT *
FROM FilteredCandidates)
GROUP BY Nation

```

In the previous example, all the numerical values were positive. The rewriting that deals with negative values is produced in a similar way: a contribute equals to zero is assigned to *MaxAcc* (instead of *MinAcc*). For instance, if the *Account* of the third tuple in *Customer* were been -500 (instead of 500), the **SELECT** clause of *UnFilteredCandidates* would have been as follows:
SELECT *Cust*, *Nation*, **MIN**(*Account*) **AS** *MinAcc*, 0 **AS** *MaxAcc*.

The query *UnFilteredCandidates* obtains the bounds for the customers that are not filtered, and therefore contribute to both bounds. The query *FilteredCandidates* obtains the bounds for the customers that are filtered, and therefore may contribute zero to some of the two bounds.

In [46] the rewriting approach above has been generalized for obtaining range-consistent query answers for queries with aggregate expressions (**MAX**, **MIN**, **SUM**) and grouping w.r.t. a set of primary key functional dependencies.

3.5 A Class of Tractable but not Rewritable Queries

By means of the query rewriting approach several tractable cases for the problem of computing consistent answers have been proposed. Specifically, the rewriting method in [4] presented in Section 2.5.2 works for quantifier-free conjunction of literals queries and (generic) binary acyclic universal constraints. As seen in Section 3.2.1, in [29] it has been shown query rewriting still works for simple conjunctive queries and one functional dependencies per relation. Further, in [45] and [46] (cfr. Section 3.3 and Section 3.4) a query rewriting for the class conjunctive tree queries and constraints consisting of at most one key dependency per relation has been proposed.

But tractable cases of the consistent query answer problem are not restricted to computing query rewriting, as seen in Section 3.2.1. More importantly, for some tractable conjunctive queries Q , query rewriting is not feasible, since (as proved in [44]) there is no first-order query Q' that can retrieve the consistent answers for Q , even thought limiting the set of integrity constraints to only key dependencies.

In [44] a polynomial-time algorithm for answering queries belonging to a class of non-rewritable query has been proposed. It is assumed that the set of integrity constraints consist of at most one key dependency per relation of the database scheme. Whereas, the queries are conjunctive query with only binary predicated and built-in predicates consisting of equality ($=$) and inequality (\neq) atoms.

We first provide an example of tractable but not rewritable query.

Example 3.13 Consider the relation scheme $P(\textit{Employee}, \textit{Salary})$ and the key dependency $\textit{Employee} \rightarrow \textit{Salary}$. Two relation instances of P , I_1 and I_2 are shown in Fig. 3.9. Note that I_2 which differs from I_1 in only the last tuple.

<i>Employee</i>	<i>Salary</i>
<i>John</i>	1000
<i>John</i>	2000
<i>Mary</i>	1000
<i>Mary</i>	2000
<i>Anna</i>	1000
<i>Anna</i>	3000

I_1

<i>Employee</i>	<i>Salary</i>
<i>John</i>	1000
<i>John</i>	2000
<i>Mary</i>	1000
<i>Mary</i>	2000
<i>Anna</i>	1000
<i>Anna</i>	2000

I_2

Fig. 3.9. Two relation instances of $P(\textit{Employee}, \textit{Salary})$

Let Q_1 be the following boolean query, which checks whether there are two (distinct) employees that have the same salary

$$Q_1 = \exists x_1, x_2, y P(x_1, y) \wedge P(x_2, y) \wedge x_1 \neq x_2$$

The consistent answer to Q_1 in the instance I_1 is *false*, since for the repair

$$R_1 = \{\langle \textit{John}, 1000 \rangle, \langle \textit{Mary}, 1000 \rangle, \langle \textit{Anna}, 3000 \rangle\}$$

the query is *false*. Whereas, consistent answer to Q_1 in the instance I_2 is *true* (it is easy to verify that for all repairs of I_2 , Q_1 is *true*).

As will be clear in the following, there is no first-order rewritten query Q'_1 that can retrieve the consistent answers for Q_1 . □

In order to find the consistent answers for Q_1 , we construct a graph G_I for the relation instance I of the (binary) scheme $P(\textit{Employee}, \textit{Salary})$. The graph G_I is a bipartite graph, with partitions $\textit{Employee}$ and \textit{Salary} . These partitions have one vertex for each value in the active domain of attributes $\textit{Employee}$ and \textit{Salary} , respectively. The set of edges of G_I consists of all tuples $\langle a, b \rangle$ of an instance of I .

It can be verified that, given a (possible inconsistent) relation instance I of a scheme $P(\textit{Employee}, \textit{Salary})$ with the $\textit{pkey}(P) = \textit{Employee}$, the consistent answer to the query Q_1 in I is *false* if and only if the graph G_I has a perfect matching.

Example 3.14 Consider the relation instances I_1 and I_2 in Fig. 3.9. A perfect matching for the graph G_{I_1} consists of the edges $\{\langle \textit{John}, 1000 \rangle, \langle \textit{Mary}, 1000 \rangle, \langle \textit{Anna}, 3000 \rangle\}$, which corresponds to the repair R_1 of the Example 3.13.

Whereas, for the instance I_2 there is no perfect matching since the cardinality of the active domain of *Employee* is greater than the cardinality of the active domain of *Salary*. Thus the consistent answer to Q_1 in I_2 w.r.t. the key dependency defined is *true*. \square

There are a number of polynomial-time algorithms in the literature for deciding the existence of a perfect bipartite matching. Therefore Q_1 is a tractable query.

However, we now show that no approach based on query-rewriting, such as the one of [4], works for Q_1 . This follows from the fact that perfect matching is not first order definable. Assume, toward a contradiction, that there is a first-order query Q'_1 such that evaluating Q'_1 in an instance I returns exactly the consistent answer to Q_1 in I . Let A_i be the set of vertices of the partition *Salary* that are neighbors of vertex v_i of the partition *Employee* in the graph G_I , for an instance I of the scheme $P(\textit{Employee}, \textit{Salary})$. A *system of distinct representatives* of A_1, \dots, A_n is a sequence of n distinct elements a_1, \dots, a_n with $a_i \in A_i$ ($1 \leq i \leq n$). Clearly, G_I has a perfect matching if and only if A_1, \dots, A_n has a system of distinct representatives. Since the consistent answer to Q_1 in I is *false* if and only if G_I has a perfect matching, then answer to the rewritten query Q'_1 in I is *false* if and only if A_1, \dots, A_n has a system of distinct representatives. But this is a contradiction, since it has been proved in [61] that relational algebra, with an appropriate encoding of sets, cannot test whether a family of sets has a system of distinct representatives.

3.5.1 On the Class of Tractable Queries

Consider a database scheme \mathcal{D} that contains binary relations P_1, \dots, P_n . Assume that the set of integrity constraints consists of n primary key dependencies, one per relation. The class of queries identified in [44] for which consistent answers can be computed in polynomial time consists of the queries having the form:

$$Q = Q_{free} \wedge Q_{exists}^1 \wedge \dots \wedge Q_{exists}^m$$

where Q_{free} is a quantifier-free query, and each Q_{exists}^l ($1 \leq l \leq m$) is a boolean query on relation P_l of the form

$$Q_{k,j_1,\dots,j_k} = \exists y_1, \dots, y_k \ Q_{j_1} \wedge \dots \wedge Q_{j_k} \wedge \bigwedge_{s \neq t} y_s \neq y_t$$

where

$$Q_{j_i} = \exists x_{i,1}, \dots, x_{i,j_i} \bigwedge_{1 \leq j \leq j_i} P_l(x_{i,j}, y_i) \wedge \bigwedge_{v \neq w} x_{i,v} \neq x_{i,w}$$

Observe that the query $Q_1 = \exists x_1, x_2, y \ P(x_1, y) \wedge P(x_2, y) \wedge x_1 \neq x_2$ is an instance of this class of query, where $Q_{free} = \emptyset$, $l = 1$, $k = 1$, $j_1 = 2$. Therefore, no approach based on first-order query-rewriting works for this class of queries.

Example 3.15 Consider the query Q_2 posed on the relation $P(\textit{Employee}, \textit{Salary})$ (having as key the attribute *Employee*).

$$Q_2 = \exists x_1, x_2, x_3, x_4, x_5, y_1, y_2 \left[\begin{array}{l} P(x_1, y_1) \wedge P(x_2, y_1) \wedge P(x_3, y_1) \wedge \\ x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3 \wedge \\ P(x_4, y_2) \wedge P(x_5, y_2) \wedge x_4 \neq x_5 \wedge \\ y_1 \neq y_2 \end{array} \right]$$

The query Q_2 checks whether there are three (distinct) employees that have the same salary y_1 and there other two (distinct) employees that have the same salary y_2 , such that y_1 is not equal to y_2 . Q_2 is also an instance of the class of query defined above, with $Q_{free} = \emptyset$, $l = 1$, $k = 2$, $j_1 = 3$ and $j_2 = 2$. □

Since the existentially-quantified subqueries Q_{exists}^l (with $1 \leq l \leq m$) do not share variables, they can be treated independently of each other. The polynomial-time algorithm for the evaluation of a single Q_{exists}^l solves instances of the *degree-constrained subgraph problem* [43], which is a generalization of perfect matching problem.

The results for the boolean queries can be extended to queries with free variables as follows. Let Q be a query without free variables. Let Q' be the query Q , where one of the existential quantifications has been removed. Therefore, Q' has a free variable x . We can evaluate Q' by instantiating it with each value of free variable x . In this way, we have to evaluate n queries, where n is the cardinality of the active domain of x . However, under the assumption of a constant number of conflicts per key, each of the instantiated queries may be much easier to compute than the original one.

The predicates P_l in a query belonging to the class above are binary predicate. If we consider ternary predicates the consistent query answer problem becomes intractable. Consider the relation scheme $P(\textit{Employee}, \textit{Manager}, \textit{Salary})$ and the key dependency $\textit{Employee} \rightarrow \{\textit{Manager}, \textit{Salary}\}$. In [29] it has been shown that obtaining the consistent answers for the query

$$Q = \exists x_1, x_2, y, z_1, z_2 P(x_1, y, z_1) \wedge P(x_2, y, z_2) \wedge z_1 \neq z_2$$

is a *coNP*-hard problem. However, we note that all the negative results in the literature apply only to specific queries; thus, they do not rule out the existence of classes of queries that are easier to compute, as that in this Section.

3.6 A Large Perspective on Repair Semantics

So far we have investigated the problem of computing consistent query answer considering two repair-semantics, namely minimal-set of inserted and deleted tuples and minimal-set of (only) deleted tuples. Under the former semantics,

repairs for a database consist of the minimum set of inserted and deleted tuple. Whereas the latter allows only tuple insertions for restoring the consistency in a database. As already observed, these two semantics yields the same set of repairs if the only violated integrity constraints are denial constraints.

In [23] the complexity of query answering has been studied considering six different assumptions on the data, which give raise six corresponding repairs-semantics. First, *strictly-exact*, *strictly-sound*, and *strictly-complete* semantics have been considered. Under the *strictly-exact* semantics, the interpretation of each relation P exactly corresponds to the extension of P in the database instance. Whereas, under *strictly-sound* semantics (resp. *strictly-complete* semantics) the interpretation of each relation P can be considered as a superset (resp. subset) of the extension of P in the database instance. Note that, the *strictly-exact*, *strictly-sound*, and *strictly-complete* semantics do not impose any minimality conditions on repairs. The sound semantics requires that a repair be a superset of the database; the exact semantics requires that it be equal to the database; and the complete semantics that it be a subset of the database.

Then, *loosely-exact*, *loosely-sound*, and *loosely-complete* semantics have been considered. Under these semantics, among all possible databases satisfying the integrity constraints, only the ones that are “as close as possible” to the actual database instance is selected as interpretation of the database. Under those semantics, repairs are constructed by adding tuples as well as by deleting them. The notion of repair under the loosely-exact semantics is identical to the notion proposed in [4] which is the first introduced in this dissertation (cfr. Section 2.5). The loosely-complete semantics does not impose the requirement that the set of deleted tuples be minimal in a repair, and loosely-sound semantics requires only that the set of deleted tuples be minimized.

In [23] it has been addressed the problem of query answering under the above repair-semantics. It is assumed that at most one key dependencies per relation and inclusion dependencies are specified for the database scheme.

3.6.1 Query Answering under Strict Repair Semantics

Let D be a database instance over the scheme \mathcal{D} and IC be a set of integrity constraints. The strictly-sound, strictly-complete and strictly-exact repair-semantics for \mathcal{D} w.r.t. IC are defined as the set of databases instance D' which are consistent w.r.t. IC ($D' \models IC$) and such that the assumptions on \mathcal{D} are satisfied, i.e.

- $Facts(D') \supseteq Facts(D)$ for the *strictly-sound* semantics;
- $Facts(D') \subseteq Facts(D)$ for the *strictly-complete* semantics;
- $Facts(D') = Facts(D)$ for the *strictly-exact* semantics.

These semantics give raise three notions of repairs: each D' is a repair under the considered semantics. Observe that, in this setting no minimality conditions are imposed on repairs.

Example 3.16 Considering the database scheme consisting of the two relations $Player(PName, Team)$ and $Team(Name, City)$. Assume that the set of integrity constraints consists of only the inclusion dependency $Player[Team] \subseteq Team[Name]$, stating that every player is enrolled in a team of a city. Consider the following (inconsistent) database instance D

$$D = \{Player(a, b), Player(a, d), Player(e, f), Team(b, c)\}$$

The set of repairs under the strictly-exact semantics is empty, since D is inconsistent (there are not two tuples in $Team$ having d and f as first component). This implies that query answering is meaningless, since every possible fact is a logical consequence of the database and the set of constraints.

The set of repairs under the strictly-complete semantics consists of the following three repairs:

$$\begin{aligned} R_1 &= \{Player(a, b), Team(b, c)\} \\ R_2 &= \{Team(b, c)\} \\ R_3 &= \emptyset \end{aligned}$$

Whereas under strictly-sound semantics, the set of repairs consists of all the database instance that can be obtained by adding (among others) at least one fact of the form $Team(d, \alpha)$ and one fact of the form $Team(f, \beta)$, where α and β are values of the database domain. □

Let D be a database instance over the scheme \mathcal{D} , KD a set of primary key functional dependencies, and ID a set of inclusion dependencies defined on \mathcal{D} . As illustrated in the example above, if the data are considered complete, then the empty database always belongs to set of repairs of D w.r.t $KD \cup ID$, independently of the constraints. Therefore, for any query Q and for any tuple t , deciding whether t belongs to every repair of D is trivial: it is always *false*.

Whereas, under strictly-exact semantics, we have two cases:

- i) D satisfies both KD and ID , therefore there is only the repair R such that $Facts(R) = Facts(D)$, and the set of consistent answers is obtained simply evaluating the query on the original database;
- ii) D violates either KD or ID , therefore there is no repair for D , and the set of consistent answers to a query Q consists of all the tuples of the same arity of Q .

The strictly-sound semantics is suitable if the set of constraints consists of inclusion dependencies only: in the presence of functional dependencies, the set of repairs may be empty (this is because the violations of functional dependencies cannot be fixed by tuple insertions). Given a database D satisfying

KD , the set of repair for D w.r.t. $KD \cup ID$ is constituted in general by several (possibly infinite) database instances, and each of them may have infinite size since there are several ways of adding facts to D . Solving a violation of an inclusion dependency by adding new tuples may lead to new violations of other dependencies, and thus there is no clear upper bound on the size of a repair, under the strictly-sound semantics. Indeed, in [23] it has been proved that the problem of consistent query answers with inclusion dependencies is undecidable under strictly-sound semantics, even if the database is consistent w.r.t. the set of primary key dependencies.

However, decidable cases have been identified, when *non-key conflicting* inclusion dependencies are considered. Let $KD \cup ID$ be a set of constraints consisting of a set of primary key functional dependencies KD and a set of inclusion dependencies ID . Then an inclusion dependencies $P[A] \subseteq Q[B]$ is a *non-key conflicting* inclusion dependencies w.r.t. KD if either: (i) no primary key dependencies are defined on Q , or (ii) B is not a strict superset of the (primary) key of Q , i.e. $pKey(Q) \not\subseteq B$. A database scheme \mathcal{D} is non-key conflicting if all the inclusion dependencies defined on \mathcal{D} are non-key conflicting (w.r.t. the set of primary key dependencies defined on \mathcal{D}).

Observe that, the class of non-key conflicting inclusion dependencies comprises the class of primary foreign key constraints.

The most relevant property of non-key conflicting inclusion dependencies is that they do not interfere with the key dependencies, so that it is possible to operate with these inclusion dependencies just as if the key dependencies were not defined in the scheme. This property entails that query answering can be solved exploiting the technique in [54], where the problem of conjunctive query containment in a database in presence of functional and inclusion dependencies has been addressed.

In [23] it has been proved that given a database D over a non-key conflicting database scheme \mathcal{D} , and a query Q consisting of union of conjunctive queries, the problem of establishing whether a tuple t is a consistent answer for Q is in *PTIME* in data complexity (under strictly-sound semantics).

3.6.2 Query Answering under Loose Repair Semantics

In the cases of strictly-exact and strictly-sound semantics, the violation of a single functional dependency may leads to the non-interesting cases in which the set of repairs is empty. Therefore, less strict assumptions on data are adopted leading to the notions of loosely-complete, loosely-sound and loosely-exact semantics.

The semantics of a database scheme \mathcal{D} with integrity constraints IC is characterized considering the set of database instances D such that: (i) D satisfies the integrity constraints IC , and (ii) approximate “at best” the satisfaction of the assumptions on \mathcal{D} .

Given a (possibly inconsistent) database D and a set of integrity constraints IC , we define an ordering on the set of all databases instance which are consistent w.r.t IC . If D_1 and D_2 are two such databases, we say that D_1 is *better* than D_2 , denoted $D_1 \gg D_2$ if:

- $\{Facts(D_1) \cap Facts(D)\} \supset \{Facts(D_2) \cap Facts(D)\}$ for the *sound* assumption;
- $\{Facts(D_1) - Facts(D)\} \subset \{Facts(D_2) - Facts(D)\}$ for the *complete* assumption;
- at least one of the following conditions holds for the *exact* assumption:
 - i) $\{Facts(D_1) \cap Facts(D)\} \supset \{Facts(D_2) \cap Facts(D)\}$ and $\{Facts(D_1) - Facts(D)\} \subseteq \{Facts(D_2) - Facts(D)\}$;
 - ii) $\{Facts(D_1) \cap Facts(D)\} \supseteq \{Facts(D_2) \cap Facts(D)\}$ and $\{Facts(D_1) - Facts(D)\} \subset \{Facts(D_2) - Facts(D)\}$

Given a (possibly inconsistent) database D and a set of integrity constraints IC , the set of repairs of D w.r.t. IC under the loosely-sound semantics (similarly for loosely-complete and loosely-exact semantics) consists of all database instances R that are maximal respect to \gg , i.e. for no other consistent database D' we have that $D' \gg R$.

Under those semantics, repairs are constructed by adding tuples as well as by deleting them. Specifically, under *loosely-sound* semantics the set of repairs consists of databases satisfying the integrity constraints and such that are “as sound as possible”, thus only consistent databases that minimize elimination of facts from D must be considered, independently of the set of facts added to D because completeness can be achieved without restriction on tuples to be inserted into D .

Under *loosely-complete* semantics the set of repairs consists of consistent databases that minimize the set of facts added to D (independently of the set of deleted facts). Observe that, every repair R belonging to the set of repair obtained under strictly-complete semantics is *better* than any other repair R' obtained adding facts to R , i.e. $R \gg R'$ holds. Thus, the strictly-complete semantics and the loosely-complete semantics always coincide.

The notion of repair under the *loosely-exact* semantics is identical to the notion proposed in [4], i.e., the set of facts which are either added to or deleted from the original instance are minimized (cfr. Section 2.5).

Example 3.17 Considering the database scheme consisting of the two relations $Player(PName, Team)$ and $Team(Name, City)$, as in Example 3.16. Assume that the set of integrity constraints consists of the inclusion dependency of Example 3.16 $Player[Team] \subseteq Team[Name]$, plus the key dependency $Pname \rightarrow Team$. Consider the same instance of Example 3.16:

$$D = \{Player(a, b), Player(a, d), Player(e, f), Team(b, c)\}$$

Observe that, under strictly-sound or strictly-exact semantics there are no repair for D since there a violation of the functional dependency.

On the other hand, the set of repairs under the loosely-exact semantics consists of the database instance $R_1 = \{Player(a, b), Team(b, c)\}$ and all the instances of the form:

$$\begin{aligned} R_2 &= \{Player(a, d), Team(b, c), Team(d, \alpha)\} \\ R_3 &= \{Player(a, b), Player(e, f), Team(b, c), Team(f, \alpha)\} \\ R_4 &= \{Player(a, d), Player(e, f), Team(b, c), Team(d, \alpha), Team(f, \beta)\} \end{aligned}$$

where α and β are values of the database domain.

Under loosely-sound semantics, the set of repairs consists of all the database instance of the form R_3 and R_4 , and each consistent database that can be obtained by adding facts to a database instance of the form R_3 or R_4 . The databases of the form R_2 are not maximal w.r.t. the order \gg because there is a database of the form R_4 such that $R_4 \gg R_2$, i.e R_4 is obtained deleting a subset of the facts deleted by R_2 . Similarly, for each repair R' obtained by adding facts to R_2 , there is a repair R'' obtained by adding facts to R_4 such that $R'' \gg R'$.

As for the strictly-complete semantics, the set of repairs under the loosely-complete semantics consists of the following three repairs:

$$\begin{aligned} R_1 &= \{Player(a, b), Team(b, c)\} \\ R_2 &= \{Team(b, c)\} \\ R_3 &= \emptyset \end{aligned}$$

□

In [23] the problem of computing consistent answers to queries under the loose semantics has been studied. In particular, the authors focus on query answering under the loosely-sound and the loosely-exact semantics, since the loosely-complete and the strictly-complete semantics coincide.

It has been shown that for general key dependencies and inclusion dependencies the problem of consistent query answers under loosely-sound and loosely-exact semantics is undecidable. The decidable cases identified in [23] involve again *non-key-conflicting* inclusion dependencies: given a database D over non-key conflicting database scheme \mathcal{D} , and a query Q consisting of union of conjunctive queries, the problem of establishing whether a tuple t is a consistent answer for Q is *coNP*-complete (under loosely-sound semantics) and Π_2^P -complete (under loosely-exact semantics).

Let D be a database instance over the scheme \mathcal{D} , KD be a set of primary key functional dependencies, and ID be a set of inclusion dependencies defined on \mathcal{D} . If D satisfies KD ($D \models KD$), then the problem of consistent query answers is undecidable under loosely-sound semantics, but is in *P*TIME under loosely-exact semantics. The latter result has been established by means of the algorithm shown in Fig. 3.10. The database D_1 computed by the algorithm is a repair for D w.r.t. $KD \cup ID$. Moreover, for any other repair R for D w.r.t. $KD \cup ID$, $D_1 \subseteq R$ holds. Since the condition tested at the Step 09)

corresponds to standard query answering over a relational database (D_1 in this case) the algorithm works in polynomial time.

Observe that the strategy presented above can be still adopted for computing consistent answers in databases w.r.t. a set of only inclusion dependencies. Indeed, the assumption that D satisfies KD can be replaced by that where there are no key dependencies defined for D (thus there are no key-conflicting values).

INPUT:

KD : set primary key functional dependencies defined on \mathcal{D}

ID : set of inclusion dependencies defined on \mathcal{D}

D : database instance of \mathcal{D} such that $D \models KD$

Q : conjunctive query of arity n

t : n -tuple

OUTPUT:

true if t is a consistent answer to Q in D , *false* otherwise

VAR:

D_0, D_1 : database instances of \mathcal{D}

begin

01) $D_1 = D$

02) **repeat**

03) $D_0 = D_1$

04) **for each** $P(t') \in D_1$

05) **if there exists** $P[A_1, \dots, A_k] \subseteq S[B_1, \dots, B_k] \in ID$ **such that**

06) **for each** $S(t'') \in D_1, t'[A_1, \dots, A_k] \neq t''[B_1, \dots, B_k]$

07) **then** $D_1 = D_1 - \{P(t')\}$

08) **until** $D_1 = D_0$

09) **if** $t \in D_1$ **then return** *true*

10) **else return** *false*

end

Fig. 3.10. Algorithm for computing consistent query answer on key-consistent databases

Some of the complexity results obtained in [23] for the problem of computing consistent answers to conjunctive queries are reported in Fig. 3.11. There are two tables that present, respectively, the complexity of query answering to conjunctive queries for the class of *general* database instances and for database instances which are consistent w.r.t. primary key functional dependencies.

We now compare the six repair semantics presented in this Section and that of [29] introduced in Section 3.2.1.

Under the (maximal) complete semantics in [29], repairs are obtained by deleting tuples so that a maximal consistent database is obtained from the original one. Under the strictly-exact, strictly-sound, and strictly-complete

Data complexity for *general* database instances:

Set of constraints		Semantics		
KD	ID	strictly-sound	loosely-sound	loosely-exact
no	general	$PTIME$	$PTIME$	$PTIME$
yes	non-key conflicting	$PTIME$	coNP-complete	Π_2^P -complete
yes	general	undecidable	undecidable	undecidable

Data complexity for *key-consistent* database instances:

Set of constraints		Semantics		
KD	ID	strictly-sound	loosely-sound	loosely-exact
no	general	$PTIME$	$PTIME$	$PTIME$
yes	non-key conflicting	$PTIME$	$PTIME$	$PTIME$
yes	general	undecidable	undecidable	$PTIME$

Fig. 3.11. Complexity of CQA for key functional dependencies and inclusion dependencies

semantics repairs are obtained performing, respectively, no actions, only insertions and only deletions of tuples in the original instance. But, no (partial) order among these repairs is considered, i.e. they are all minimal ones. On the other hand, under loosely-exact, loosely-sound, and loosely-complete semantics order among repairs is defined. We have already observed that the set of repairs under the *loosely-exact* semantics is exactly that obtained under the notion of repair proposed in [4] (presented in Section 2.5), which differs from that of [29]. Moreover, under loosely-sound semantics the partial order among repairs is introduced only for selecting repairs such that the set of deleted tuples is minimized, independently of the set of tuples added to the original database. Similarly, under loosely-complete semantics the set of added tuples is required to be minimal, independently of the deleted tuples.

Thus, it is clear the none of the repair semantics presented in this Section coincide to that introduced in Section 3.2.1.

3.7 Discussion

In this chapter we have investigated the computational complexity of both the repair checking and the consistent query answer problem among several dimensions. We have distinguished the following parameters. As first parameter we have considered *the repair-semantics*. The semantics proposed in [4] (and presented in Section 2.5) requires that the symmetric difference between a database and its repair be minimized; we have also discussed the semantics introduced in [29] (cfr. Section 3.2.1) and the relationship between this semantics and that of [4]; moreover, we have examined the several forms of

semantics discussed in [23] (presented in Section 3.6), and we have compared them with the two semantics above.

As second parameter, we have distinguished the *class of queries* to which belongs the query posed on the (possible inconsistent) database. We have considered different subclasses of first-order queries, i.e. quantifier-free queries (cfr. Section 3.2.1), conjunctive queries (cfr. Section 3.6.2) and their subclass such as simple conjunctive queries and tree queries (cfr. Section 3.3 and Section 3.4). Moreover, in Section 3.1 and Section 3.4.2 we have studied the consistent query answer problem for aggregation queries. In this context, the notion of consistent query answer has been revisited yielding the notion of *range-consistent query answer*, which results appropriate for this type of queries.

As third parameter, we have distinguished the *class of integrity constraints* which are defined on the (possible inconsistent) database. We have considered arbitrary denial constraints in Section 3.2.1, functional dependencies in Section 3.1, key constraints (at most one per relation) in Section 3.3, Section 3.4 and Section 3.5. Moreover, we have discussed the consistent query answer problem in databases in presence of inclusion dependencies in Section 3.2.2 and in Section 3.6. In both cases we have also studied the interactions of inclusion dependencies with key dependencies, but under different repair-semantics.

In order to define semantics for aggregation queries, *range-consistent query answer* has been introduced in [7, 8], and queries with *scalar* and *aggregation* functions have been distinguished. The former ones return a single value for the entire relation, whereas the latter ones perform grouping on a set of attributes and return a single value for each group. In [7, 8] (cfr. Section 3.1) the complexity of range-consistent query answers has been studied for scalar aggregation queries (with respect to the functions $\text{MIN}(A)$, $\text{MAX}(A)$, $\text{COUNT}(A)$, $\text{SUM}(A)$, $\text{AVG}(A)$ or $\text{COUNT}(\ast)$). A compact representation of repair, namely the *conflict graph* (cfr. Section 3.1.1), was exploited in order to characterize the complexity of computing the *greatest-lower-bound* and the *least-upper-bound answers* (cfr. Section 3.1.2).

Aggregation queries with grouping has been subsequently investigated in [46] (cfr. Section 3.4.2). The notion of range-consistent answers has been extended in such a way that for every consistent answer to the group query (where all the aggregate expression are removed) and for each aggregation expression, a range-consistent answer is provided. In order to compute such answers a rewriting algorithm for SQL queries has been proposed.

In Section 3.1.2 and in Section 3.4.2 query answering with respect to functional dependencies and key dependencies, respectively, has been studied. These class of constraints are special subclass of denial constraints. In this chapter we observed that for denial constraints, the set of consistent query answers is not affected by the choice of the primitives adopted for restoring the consistency, i.e. either insertions/deletions of tuples or only deletions of tuples. Thus, the two repair semantics presented in Section 2.5 and Section 3.2.1 coincide for denial constraints.

In Section 3.2.1 several classes of queries and constraints for which both repair checking and consistent query answer problems are in *PTIME* have been characterized. First, the tractability of the problem of computing consistent answers for quantifier-free queries and (arbitrary) denial constraints has been proved, by exploiting a generalization of the conflict graph, namely the *conflict hypergraph*. This result is an extension of that implied by the rewriting method in [4] (cfr. Section 2.5.2) for denial constraints. Second, tractability has been also provided for boolean simple conjunctive queries and a set of functional dependencies consisting of at most one dependency per relation. Then, the positive results above has been used for identifying tractable case of consistent answering for queries that are quantifier-free or simple conjunctive, and constraints that are key functional dependencies and foreign key constraints (with at most one key per relation) (cfr. Section 3.2.2). In fact the interaction between the two type of dependencies in the set of integrity constraints is limited, in such a way that the polynomial-time results presented for functional dependencies only can be exploited. Moreover, in presence of inclusion dependencies the assumption of working with complete data is a vantage, since entails that facts in every repair are a subset of the facts present in the original database instance.

Several classes of queries and constraints for which the problem of computing consistent answer is *coNP*-complete have been shown in [29] (cfr. Section 3.2.1). Specifically, intractability for boolean conjunctive queries and for functional dependencies has been shown. This result still hold when key dependencies and conjunctive queries without repeated relation symbols are considered. In [45] for a subclass of conjunctive queries without repeated relation symbols and key dependencies, an algorithm that produces a first-order query rewriting for the problem of computing consistent answers has been proposed (cfr. Section 3.3). The algorithm works for a subclass of conjunctive queries which is defined in terms of the *join graph* of the query. The join graph is a directed graph such that: its vertices are the literals of the query; and it has an edge for each join in the query that involves some variable at the position of a non-key attribute. The algorithm works for conjunctive queries without repeated relation symbols whose join graph is a forest.

In [46] the rewriting algorithm presented in [45] has been improved and extended for working on Select-Projection-Join SQL queries (cfr. Section 3.4.1). In principle, first-order queries can be translated into SQL queries. However, the queries produced by means of the algorithm in [45] have a high level of nesting (proportional to the number of relations in the query) and are therefore very inefficient. In [46] a rewriting algorithm producing SQL queries with at most one level of nesting has been proposed. The system ConQuer [46, 47] implements this rewriting algorithm.

The rewriting technique is a suitable method for obtaining consistent answers, but, as shown in [44] (cfr. Section 3.5), there are class of tractable first-order queries for which no rewriting providing consistent answers exists in presence of only key constraints. In order to obtain a polynomial-time algo-

rithm for computing consistent answers for this class of non-rewritable queries, an approach based on graph theory has been proposed.

The complexity of the consistent query answer problem in presence of both functional and inclusion dependencies has been studied in [23] (cfr. Section 3.6) and in [29] (cfr. Section 3.2.2), but under different repair-semantics. In the former, under *loosely-exact* semantics a repair is obtained performing a minimal set of insertions and deletions of tuples, whereas in the latter only deletions are possible. In [23] it has been shown that for key dependencies and inclusion dependencies the consistent query answers problem is undecidable. Decidable cases have been identified limiting the interaction between the two type of dependencies. Specifically, either introducing *non-key-conflicting* inclusion dependencies for *general* databases, or considering *general* inclusion dependencies for *key consistent* databases. On the other hand, by forcing the repairs to be subsets of the original database, as in [29], makes the problem of consistent query answers decidable for *general* database instances and *general* inclusion dependencies.

Logic Programs and Database Repairs

Repairs can be specified using logic programs with disjunction and classical negation. In some approaches in literature, repairs are represented by *answer sets* of such logic programs [5, 9, 12, 51, 52]. In this case consistent query answers can be obtained by skeptical answer set semantics for the achieved logic program, i.e. computing facts true in every answer set. This is a very general approach that can handle arbitrary first-order queries and universal integrity constraints. On the other hand, for the class of logic program used the computational complexity of skeptical reasoning is Π_2^P -complete (in data complexity). Thus, in general, the application of these approaches is expensive.

In this chapter we present some techniques that exploit the expressive power of (*extended*) *disjunctive logic programs* (with both *classical* negation and *negation as failure*) for specifying repairs for a database.

4.1 Logic Programs

Before describing how logic programs are used for specifying repair, we briefly recall some notions on logic programs. A comprehensive discussion can be found in [10].

4.1.1 General Logic Programs

The language of a logic program, like a first-order language, is determined by its constants, function and predicate symbols. Terms are built as in the corresponding first-order language; atoms have the form $P(e_1, \dots, e_n)$, where e_1, \dots, e_n are terms and P is an n -ary predicate symbol.

A *rule* (or *clause*) is an expression of the form

$$\alpha_0 \leftarrow \alpha_1, \dots, \alpha_m, \text{not } \alpha_{m+1}, \dots, \text{not } \alpha_n$$

where each α_i is an atom and *not* is the logical connective called *negation as failure* [32, 70]. The left-hand side of the rule is called the *head* (or *conclusion*)

of the rule; the right-hand side is called the *body* (or *premise*) of the rule. A collection of rules is called a *general logic program* (also said a *normal logic program*). General logic programs that do not have *not* are called *definite programs*. The Herbrand Base of a program Π is the set of all ground atoms in the language of Π .

A logic program can be viewed as a specification for building possible theories of the world, and the rules can be viewed as constraints these theories should satisfy. Semantics of logic programs differ in the way they define satisfiability of the rules. Under *stable model semantics* [49] the corresponding theories are sets of ground atoms, called the *stable models* of a program.

The *stable model* of a definite program Π is the smallest subset S of the Herbrand Base such that for any rule $\alpha_0 \leftarrow \alpha_1, \dots, \alpha_m$ from Π , if $\alpha_1, \dots, \alpha_m \in S$ then $\alpha_0 \in S$.

Let Π be an arbitrary general logic program. For any set S of atoms, let Π^S the program obtained from Π by deleting

- i) each rule that has a formula *not* α in its body with $\alpha \in S$, and
- ii) all formulas of the form *not* α in the body of the remaining rules.

Clearly, Π^S is a definite program (it does not contain *not*), so that its stable model is already defined. If this stable model coincides with S , then we say that S is a stable model of Π .

A ground atom $P(t)$ is *true* in S if $P(t) \in S$, otherwise $P(t)$ is *false* (i.e. $\neg P(t)$ is *true*) in S . The definition is extended to arbitrary first-order formulas in the standard way. A program Π implies a formula φ ($\Pi \models \varphi$) if φ is *true* in all stable models of Π . An answer to a ground query $Q(t)$ is *yes* if $\Pi \models Q(t)$, *no* if $\Pi \models \neg Q(t)$ and *unknown* otherwise.

Programs which have a unique stable model are called *categorical*.

4.1.2 Extended Logic Programs

In addition to negation-as-failure *not*, “extended” logic programs contains a second form of negation \neg , called *classical* or *strong* negation. General logic programs provide negative information implicitly, through closed-world reasoning. Whereas, an extended logic program can include explicit negative information. In the language of extended programs, we can distinguish between a query which fails in the sense that it *does not succeed* and a query which fails in the stronger sense that its *negation succeeds*.

An *extended logic program* (ELP) is a collection of rules of the form

$$L_0 \leftarrow L_1, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n$$

where each L_i is a literals, i.e. formulas of the form $P(u)$ or $\neg P(u)$, where $P(u)$ is an atom.

Let *Lit* be the set of ground literals in the language of the extended logic program Π . The semantics of the extended logic program Π assigns to it a

collection of its *answer sets* (set of literals corresponding to beliefs) [50]. A literal $\neg P(t)$ is *true* in an answer set S if $\neg P(t) \in S$ (recall that *not* $P(t)$ is *true* in S if $P(t) \notin S$). An answer to a ground query $Q(t)$ is *yes* if $Q(t)$ is true in all answer sets of Π , *no* if the complementary literal of $Q(t)$ is *true* in all answer sets of Π , and *unknown* otherwise.

Let Π be a program without negation-as-failure, i.e. a collection of rules of the form $L_0 \leftarrow L_1, \dots, L_m$. The unique *answer set* of Π , denoted as $as(\Pi)$, is the smallest (in the sense of set-theoretic inclusion) subset S of Lit such that:

- i) for any rule $L_0 \leftarrow L_1, \dots, L_m$ from Π , if $L_1, \dots, L_m \in S$ then $L_0 \in S$;
- ii) if S contains a pair of complementary literals, then $S = Lit$; in this case Π is said to be *contradictory*.

Now, consider a ground extended logic program Π . By Lit we again denote the set of ground literal in the language of Π . For any set $S \subseteq Lit$, let Π^S be the program obtained from Π by deleting

- i) each rule that has a formula *not* L in its body with $L \in S$, and
- ii) all formulas of the form *not* L in the body of the remaining rules.

The resulting program Π^S is a ground extended logic program without *not*, so its answer set $as(\Pi^S)$ is already defined. If $as(\Pi^S)$ coincides with S , then we say that S is an answer set of Π .

As shown in [50] the extended logic program can be further transformed into an equivalent *general logic program* with stable model semantics.

4.1.3 Extended Disjunctive Logic Programs

By an *extended disjunctive logic program* (EDLP) we mean as a collection of rules of the form

$$L_0 \vee \dots \vee L_k \leftarrow L_{k+1}, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n$$

where each L_i is a literals. When each L_i is an atom we refer to the program as a *normal disjunctive logic program*. When $n = m$ and each L_i is an atom, we refer to the program as a *positive disjunctive logic program*.

The definition of an answer set of a disjunctive logic program [50, 69] is almost identical to that of extended logic programs. Let Π be an extended disjunctive logic program without negation-as-failure. Let Lit be the set of ground literals in the language of Π . An *answer set* of Π is a smallest (in the sense of set-theoretic inclusion) subset S of Lit such that:

- i) for any rule $L_0 \vee \dots \vee L_k \leftarrow L_{k+1}, \dots, L_m$ from Π , if $L_{k+1}, \dots, L_m \in S$ then for some i , with $0 \leq i \leq k$, $L_i \in S$;
- ii) if S contains a pair of complementary literals, then $S = Lit$.

Unlike extended logic programs without *not*, an extended disjunctive logic program without *not* may have more than one answer set. We denote the answer sets of an extended disjunctive logic program Π without *not* by $ass(\Pi)$.

Now, consider an arbitrary ground extended disjunctive logic program Π . By Lit we again denote the set of ground literal in the language of Π . For any set $S \subseteq Lit$, let Π^S be the program obtained from Π by deleting

- i) each rule that has a formula *not* L in its body with $L \in S$, and
- ii) all formulas of the form *not* L in the body of the remaining rules.

The resulting program Π^S is a ground disjunctive extended logic program without *not*, so its answer sets $ass(\Pi^S)$ are already defined. If $S \in ass(\Pi^S)$, then we say that S is an answer set of Π . The answer sets of Π are the intended stable models of Π .

4.1.4 Logic Programs with Exceptions

A logic program with exception (LPE) [57] is a program with the syntax of an extended logic program (ELP), i.e. it consists of clauses of the form

$$L_0 \leftarrow L_1, \dots, L_m, \textit{not } L_{m+1}, \dots, \textit{not } L_n \quad (4.1)$$

Rules (or *defaults*) are clause with positive head, whereas the clause with negative head are said to be *exceptions*.

The semantics of an LPE is obtained from the semantics of ELPs, by adding an extra condition that assigns higher priority to exceptions (w.r.t. defaults). The answer set $as(\Pi)$ of an LPE Π consisting of rules without *not* is defined as for an ELP without *not*.

Now, consider a ground LPE Π with *not*. By Lit we again denote the set of ground literal in the language of Π . For any set $S \subseteq Lit$, let Π^S be the program obtained from Π according to the following steps:

- 1) delete every clause that has a formula *not* L in its body with $L \in S$, and
- 2) delete all formulas of the form *not* L (with $L \notin S$) in the body of the remaining clauses, and
- 3) delete every default clause having a positive conclusion L with $\neg L \in S$.

The resulting program Π^S is a ground ELP without *not*, so its answer set $as(\Pi^S)$ is already defined. We say that a set S of ground literals is an *e-answer set* for the original LPE Π if $S = as(\Pi^S)$. The e-answer sets are the intended models of the original program.

Above, 1) and 2) are as in the *answer set semantics* for extended logic programs, whereas 3) gives an account of exceptions. The e-answer sets are in correspondence with answer set of extended logic programs [57]. This can be established by transforming the original LPE into an ELP with the answer set semantics, that can be further transformed into an equivalent *general logic program* with stable model semantics [50].

4.1.5 Prioritized Logic Programs

A prioritization mechanism can be applied to a general extended disjunctive program. A partial preference relation \preceq among literals is defined as follows. Given two literals L_1 and L_2 , $L_1 \preceq L_2$ means that L_2 has higher *priority* than L_1 , that is, for each e_1 instance of L_1 and for each e_2 instance of L_2 , it holds $e_1 \preceq e_2$ (clearly, the sets of ground instantiation of L_1 and L_2 must have empty intersection). Moreover, if $L_1 \preceq L_2$ and $L_2 \preceq L_3$, then $L_1 \preceq L_3$.

A *prioritized logic program* (PLP) is a pair $\langle \Pi, \Phi \rangle$, where Π is a standard program and Φ is a set of priorities [75, 80].

Let Φ^* be the set of priorities which can be reflexively or transitively derived from Φ . Given a prioritized logic program $\langle \Pi, \Phi \rangle$, the relation \sqsubseteq is defined over the stable models of Π as follows. For any stable model M_1 , M_2 and M_3 of Π ,

1. $M_1 \sqsubseteq M_1$;
2. $M_1 \sqsubseteq M_2$ if
 - a) $\exists L_1 \in (M_1 - M_2), \exists L_2 \in (M_2 - M_1)$ such that $L_1 \preceq L_2 \in \Phi^*$ and
 - b) $\nexists L_3 \in (M_1 - M_2)$ such that $L_2 \preceq L_3 \in \Phi^*$;
3. if $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_3$ then $M_1 \sqsubseteq M_3$.

If $M_1 \sqsubseteq M_2$ we say that M_2 is *preferable* to M_1 . An interpretation M is said to be a *preferred* stable model of $\langle \Pi, \Phi \rangle$ if there is no interpretation M' such that $M' \neq M$ and $M \sqsubseteq M'$.

4.2 Querying Databases using Logic Programs with Exceptions

In [5, 9] a generalization of *Logic Program with Exceptions* (LPE) has been exploited for specifying database repairs in order to retrieve consistent query answers. Specifically, in a (classical) LPE, rules with a positive literal in the head represent a sort of general default, whereas rules with a logically negated head represent exceptions. In [5, 9] LPE has been extended in such a way that the program will have also *negative default* and *positive exceptions* (to negative default). Moreover, also disjunctions of literals in the head of some clauses is present. This program will be a *Disjunctive Logic Program with Exceptions* (DLPE).

It was shown that, for a set of (domain independent) binary integrity constraints BC and a given database instance D there is a one-to-one correspondence between the *e-answers set* of the constructed DLPE repair program, and the repairs of D w.r.t. BC . Therefore, the consistent query answer for general first-order queries is obtained asking for the atoms which are *true* in every *e-answer set* of the logic program.

4.2.1 Extending Logic Programs with Exceptions

In order to specify database repairs, the syntax and the semantics of logic program with exceptions have been extended. In logic program with exceptions the consequences of default rules can be overridden by the consequences of exceptions rules. Logic program with exceptions are extended to contain also *negative defaults*, i.e. defaults with negative conclusions which can be overridden by *positive exceptions* (rules with a positive head, but representing exceptions to negative default). Moreover, *disjunctive exceptions* are introduced in the extension, i.e., rules of the form

$$L_1 \vee \dots \vee L_k \leftarrow L_{k+1}, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n \quad (4.2)$$

where L_i are literals. Note that, only disjunctive exception are introduced, *disjunctive defaults* are not used in the framework of [5, 9].

The programs obtained with these extensions will be called *Disjunctive Logic Program with Exceptions* (DLPE). The semantics of DLPEs is obtained by extending the e-answer semantics of LPEs as follows.

Let Π be a ground DLPE and S a set of ground literals which is candidate to be a model of Π . A pruned program Π^S is obtained performing the following steps:

- 1) delete every clause that has a formula *not* L in its body with $L \in S$, and
- 2) delete all formulas of the form *not* L (with $L \notin S$) in the body of the remaining clauses, and
- 3') delete every (positive) default having a positive conclusion A with $\neg A \in S$; and every (negative) default having a negative conclusion $\neg A$ with $A \in S$.

Observe that, Steps 1), 2) are identical to that of Section 4.1.4 defining the answer set semantics of a logic program with exceptions, whereas the Step 3') replaces the Step 3) of Section 4.1.4.

Applying 1), 2) and 3') to the ground program Π , a ground *disjunctive extended logic program* Π^S without *not* is obtained. If the candidate set S is one of the minimal models of Π^S (i.e. $S \in \text{ass}(\Pi^S)$), then we say that S is an *e-answer set* for Π .

4.2.2 Specifying Repairs

The approach considers a set of binary integrity constraints BC written in the *standard format* (cfr. Section 1.3.1), i.e. universally quantified first order formulas with at most two literals, having the form

$$\forall X_1 X_2 [L_1(X_1) \vee L_2(X_2) \vee \phi(X_1, X_2)]$$

where L_1, L_2 are literals associated to the database scheme and X_1, X_2 are tuples of variables and ϕ is a formula containing only built-in predicates and free variables appearing in L_1 and L_2 .

There are three possibilities for binary constraints in terms of sign of literals in them, namely the universal closures of:

$$(a) P_1(X_1) \vee P_2(X_2) \vee \phi(X_1, X_2)$$

$$(b) P_1(X_1) \vee \neg P_2(X_2) \vee \phi(X_1, X_2)$$

$$(c) \neg P_1(X_1) \vee \neg P_2(X_2) \vee \phi(X_1, X_2)$$

where P_1 and P_2 are relation atoms.

Given a database D a set of binary integrity constraints BC , the approach consists in the specification of the repairs for D w.r.t. BC , by means of a *Disjunctive Logic Program with Exceptions* Π^D . In Π^D for each predicate P that participates in some integrity constraint in BC , a new predicate P' representing the repaired version on P is introduced. P' contains the tuples corresponding to P in a repair of the original database.

Moreover, Π^D is obtained by introducing:

1. **Persistence Defaults.** For each base predicate P , the method introduces the persistence defaults:

$$P'(X) \leftarrow P(X) \tag{4.3}$$

$$\neg P'(X) \leftarrow \text{not } P(X) \tag{4.4}$$

The defaults say that all data persist from the original relation instances to their repaired versions.

The *positive defaults* (rules of type (4.3)) will be subject to negative exceptions, whereas the *negative defaults* (rules of type (4.4)) will be subject to positive exceptions.

Example 4.1 Consider the set of full inclusion dependencies

$$ID = \{\forall xy P(x, y) \Rightarrow Q(x, y), \forall xy Q(x, y) \Rightarrow S(x, y)\}$$

and the (inconsistent) database instance $D = \{P(a, b), Q(a, b)\}$. Translating ID in standard format we obtain

$$ID = \{\neg P(x, y) \vee Q(x, y), \neg Q(x, y) \vee S(x, y)\}$$

where the universal closure of the formulas is omitted.

In order to specify the database repairs the new predicates P' , Q' and S' are introduced. The following six default rules are generated:

$$\begin{array}{ll} P'(x, y) \leftarrow P(x, y); & \neg P'(x, y) \leftarrow \text{not } P(x, y); \\ Q'(x, y) \leftarrow Q(x, y); & \neg Q'(x, y) \leftarrow \text{not } Q(x, y); \\ S'(x, y) \leftarrow S(x, y); & \neg S'(x, y) \leftarrow \text{not } S(x, y). \end{array}$$

□

2. **Stabilizing Exceptions.** For each BC of the form (a), i.e. $P_1(X_1) \vee P_2(X_2) \vee \phi(X_1, X_2)$, the 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 a formula that is logically equivalent to the logical negation of ϕ .

Similarly, for each BC 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 BC 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.

Example 4.2 (Example 4.1 continued) The following four stabilizing exception rules are generated:

$$\begin{aligned} \neg P'(x, y) &\leftarrow \neg Q'(x, y); & Q'(x, y) &\leftarrow P'(x, y); \\ \neg Q'(x, y) &\leftarrow \neg S'(x, y); & S'(x, y) &\leftarrow Q'(x, y); \end{aligned}$$

□

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

$$(a) \quad P'_1(X_1) \vee P'_2(X_2) \leftarrow \text{not } P_1(X_1), \text{not } P_2(X_2), \varphi(X_1, X_2)$$

$$(b) \quad P_1(X_1) \vee \neg P_2(X_2) \leftarrow \text{not } P_1(X_1), P_2(X_2), \varphi(X_1, X_2)$$

$$(c) \quad \neg P_1(X_1) \vee \neg P_2(X_2) \leftarrow P_1(X_1), P_2(X_2), \varphi(X_1, X_2)$$

These rules are necessary as a first step toward the repair of D . They trigger the first changes, from the P_i s to the P'_i s; next the stabilizing exceptions propagate all required changes.

Example 4.3 (Example 4.2 continued) The following two (disjunctive) trigger exception rules are generated:

$$\begin{aligned} \neg P'(x, y) \vee Q'(x, y), \leftarrow P(x, y), \text{not } Q(x, y); \\ \neg Q'(x, y) \vee S'(x, y) \leftarrow Q(x, y), \text{not } S(x, y); \end{aligned}$$

Each of these rules represents the two possible ways to repair the corresponding integrity constraint, separately. For instance, the first rule says that in order to (locally) repair the first constraint, either eliminate (x, y) from P or insert (x, y) into Q . □

Finally, the *facts* corresponding to the original database and rules for the built-in predicates are added. The program Π^D constructed as shown above is the *repair logic program with exceptions for the database instance D*. In Π^D positive defaults are blocked by negative conclusions, and negative defaults, by positive conclusions. The semantics of Π^D is the e-answer semantics of DLPEs.

Example 4.4 (Example 4.3 continued) It is possible to verify that the e-answer sets of the program are

$$\begin{aligned} as_1 &= \{P'(a, b), Q'(a, b), S'(a, b), P(a, b), Q(a, b), \dots\} \\ as_2 &= \{\neg P'(a, b), \neg Q'(a, b), P(a, b), Q(a, b), \dots\} \end{aligned}$$

where the literals not shown explicitly are negative literals, e.g. $\neg P'(a, a)$, $\neg Q'(b, a)$ inherited from the original instance with the negative defaults. The corresponding repairs are $R_1 = \{P'(a, b), Q'(a, b), S'(a, b)\}$ and $R_2 = \{\neg P'(a, b), \neg Q'(a, b)\}$, respectively. The presence of the literal $S'(a, b)$ in R_1 represents the insertion of the tuple (a, b) in the relation S . The repair R_2 consists of the deletion of both $P'(a, b)$ and $Q'(a, b)$. □

As shown in [57] for LPEs, a DLPE having *e-answer* set semantics can be transformed into a Disjunctive Extended Logic Program (without exceptions) with answer set semantics, by transforming the persistence defaults (4.3) and (4.4), respectively, into

$$\begin{aligned} P'(x) \leftarrow P(x), \text{not } \neg P'(x) \\ \neg P'(x) \leftarrow \text{not } P(x), \text{not } P'(x) \end{aligned}$$

As shown in [50] the resulting program can be further transformed into an equivalent *disjunctive normal logic program* (i.e. without logical negation \neg) with stable model semantics [49].

4.2.3 Computing Consistent Query Answers

Consider a set of domain independent binary integrity BC , for which checking their satisfaction in an instance D can be done considering the elements of the finite active domain only [74]. Then for a database instance D , there is a one-to-one correspondence between the e-answer sets of the repair program Π^D and the set of repairs for D w.r.t. BC .

The consistent answers to a query Q in a database D is evaluated as follows. First, from Q a stratified logic program $\Pi(Q)$ is obtained (this is a standard construction [1]) in terms of new primed predicates. One of the predicate symbols, Ans_Q of $\Pi(Q)$ is designated as the query predicate: its extension is the answer to Q in D . Second, the e-answer set S_1, \dots, S_k of the program $\Pi(Q) \cup \Pi^D$ is determined (as stated above, a Disjunctive Extended Logic Program with answer set semantics can be used here). Third, the consistent answers to Q in D is the result of the intersection $\bigcap_{1 \leq i \leq k} S_i/Ans_Q$, where S_i/Ans_Q is the extension of Ans_Q in S_i . The consistent answers to a query are those that can be obtained from the repair program plus the query program under the *cautious* or *skeptical* answer set semantics for the combined logic program: what is true of the program is what is true in all answer sets.

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 [37] can be used for computing repairs and consistent query answers.

4.2.4 Referential Integrity Constraints

The approach can be extended to ternary integrity constraints in standard format, but in this case also disjunctive stabilizing rules are necessary. Moreover, when the methodology is extended to a constraint in standard format having k literals, the number of stabilizing exception rules grows according to the number of subsets of the database literals in the constraint, i.e. exponentially in k .

The methodology presented for binary integrity constraints in standard format can be applied to referential integrity constraints (non-full inclusion dependencies). This can be done via an appropriate representation of existential quantifiers as program rules.

Example 4.5 Consider the referential integrity constraint $\forall x P(x) \Rightarrow \exists y Q(x, y)$ and the inconsistent database instance $D = \{P(a), P(b), Q(b, a)\}$. We assume that there is an underlying database domain $\mathbf{dom} = \{a, b\}$. The repair program has the persistence defaults clauses

$$\begin{aligned} P'(x) &\leftarrow P(x); & \neg P'(x) &\leftarrow \text{not } P(x); \\ Q'(x, y) &\leftarrow Q(x, y); & \neg Q'(x, y) &\leftarrow \text{not } Q(x, y). \end{aligned}$$

In addition it has the stabilizing exceptions

$$\begin{aligned} \neg P'(x) &\leftarrow \neg Q'(x, \text{null}), \text{not } \text{aux}'(x); \\ Q'(x, \text{null}) &\leftarrow P'(x), \text{not } \text{aux}'(x). \end{aligned}$$

with $\text{aux}'(x) \leftarrow Q'(x, y)$. The variables of this program range over **dom**, that is, they do not take the value *null*. The literal $\text{aux}'(x)$ in the clause $Q'(x, \text{null}) \leftarrow P'(x), \text{not } \text{aux}'(x)$ is necessary to insert a null value only when it is needed. Then the literal $\neg Q'(x, \text{null})$ is used in the first stabilizing exception. Finally the program has the following trigger exception

$$\neg P'(x) \vee Q'(x, \text{null}) \leftarrow P(x), \text{not } \text{aux}(x);$$

with $\text{aux}(x) \leftarrow Q(x, y)$.

Instantiating variables on *Dom* only, the two answer sets are the expected ones, namely delete $P(a)$ or insert $Q(a, \text{null})$. □

4.3 Querying Database using Extended Disjunctive Logic Programs

In [51, 52] a general logic framework for computing repairs and consistent answers over inconsistent databases has been proposed. The technique is based on the generation of an Extended Disjunctive Logic Program (EDLP) derived from the set of integrity constraints. The disjunctive program can be used for two different purposes: to compute repairs for the database, and to produce consistent answers. Moreover, *repair constraints* and *prioritized update rules* have been introduced. The former ones are constraints which can be used to specify which repairs are feasible. The latter ones are rules which allow us to give preference to some repairs of the database with respect to others. Integrity constraints with prioritized updates can be rewritten into prioritized extended disjunctive rules.

4.3.1 Computing Database Repairs

The repairs for an inconsistent database D can be generated from the stable models of an Extended Disjunctive Logic Program which is derived by rewriting the set of integrity constraints as follows.

Let ic be an universally quantified constraint of the form

$$\forall X [\beta_1 \wedge \cdots \wedge \beta_n \wedge \varphi \Rightarrow \alpha_1 \vee \cdots \vee \alpha_m]$$

where $\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_n$ are atoms and φ is a conjunction of built-in atoms. Then, $dj(ic)$ denotes the extended disjunctive rule

$$\neg\beta'_1 \vee \dots \vee \neg\beta'_n \vee \alpha'_1 \vee \dots \vee \alpha'_m \leftarrow (\beta_1 \vee \beta'_1), \dots, (\beta_n \vee \beta'_n), \varphi, \\ (\text{not } \alpha_1 \vee \neg\alpha'_1), \dots, (\text{not } \alpha_m \vee \neg\alpha'_m)$$

where γ'_i denotes the atom derived from γ_i by replacing the predicate symbol P of γ_i with the new symbol P' , that is if γ_i is $P(u)$ then γ'_i will be $P'(u)$. The derivation of the atom $P'(u)$ (resp. $\neg P'(u)$) states that the atom $P(u)$ must be inserted into (resp. deleted from) the database.

The above disjunctive rule states that if

- i) every atom β_i ($1 \leq i \leq n$) appearing in the body of the constraint is *true* (i.e. it is in the source databases or is inserted by the repair: β_i or β'_i holds, respectively), and
- ii) every atom α_j ($1 \leq j \leq m$) appearing in the head of the constraint is *false* (i.e. it is not in the source databases or is deleted by the repair: *not* α_j or $\neg\alpha'_j$ holds, respectively), and
- iii) the constraint φ is *true*,

then to satisfy the constraint either some β_i is deleted (so $\neg\beta'_i$ holds) or some α_j is inserted (so α'_j holds).

Example 4.6 Considering the integrity constraints $ic_1 = \forall x [Q(x) \Rightarrow T(x)]$, then $dj(ic_1)$ is as follows

$$\neg Q'(x) \vee T'(x) \leftarrow (Q(x) \vee Q'(x)), (\text{not } T(x) \vee \neg T'(x))$$

Whereas, for the constraint $ic_2 = \forall x [P(x) \Rightarrow S(x) \vee Q(x)]$, $dj(ic_2)$ is as follows

$$\neg P'(x) \vee S'(x) \vee Q'(x) \leftarrow (P(x) \vee P'(x)), (\text{not } S(x) \vee \neg S'(x)), \\ (\text{not } Q(x) \vee \neg Q'(x))$$

□

Given an extended disjunctive rule r containing also body disjunctions (as the rule $dj(ic)$ obtained for a constraint ic)

$$\alpha_1 \vee \dots \vee \alpha_k \leftarrow (\beta_{1,1} \vee \dots \vee \beta_{1,m_1}), \dots, (\beta_{n,1} \vee \dots \vee \beta_{n,m_n})$$

we denote as $st(r)$ the equivalent set of (standard) disjunctive rules

$$\alpha_1 \vee \dots \vee \alpha_k \leftarrow \beta_{1,i_1} \vee \dots \vee \beta_{n,i_n} \quad \forall j, i: 1 \leq j \leq n \text{ and } 1 \leq i_j \leq m_j$$

Given a disjunctive program Π containing also rules with body disjunctions, $st(\Pi)$ denotes the (standard) disjunctive program derived from Π by rewriting body disjunctions.

Let IC be a set of universally quantified integrity constraints, then $DP(IC) = \{ dj(ic) \mid ic \in IC \}$ and $\Pi(IC) = st(DP(IC))$. Thus, $DP(IC)$

denotes the set of (generalized) disjunctive rules derived from the rewriting of IC , whereas $\Pi(IC)$ denotes the set of (standard) disjunctive rules derived from $DP(IC)$.

Given a database D and a set of constraints IC , $\Pi(IC)_D$ will denote the logic program derived from the union of the rules in $\Pi(IC)$ with the facts in D , i.e. $\Pi(IC)_D = \Pi(IC) \cup D$.

Given a database D and a set of constraints IC , every stable model M of $\Pi(IC)_D$ can be used to define a possible repair for D by interpreting new derived atoms (with primed predicates) as insertions and deletions of tuples. Thus, if a model M contains two atoms $\neg P'(t)$ and $P(t)$, we deduce that the atom $P(t)$ violates some constraint and, therefore, it must be deleted. Analogously, if M contains the derived atom $P'(t)$ and does not contain $P(t)$ (i.e. $P(t)$ is not in the database) we deduce that the atom $P(t)$ should be inserted in the database. Therefore, the set of atoms which must be deleted from the database are $U^-(M) = \{P(t) \mid \neg P'(t) \in M \wedge P(t) \in D\}$, whereas the set of atoms which must be inserted into the database D are $U^+(M) = \{P(t) \mid P'(t) \in M \wedge P(t) \notin D\}$. Given a stable model M of $\Pi(IC)_D$, the sets of update operation $U^-(M)$ and $U^+(M)$ define a repair $R(M, D)$ for D obtained from the application of $U^-(M)$ and $U^+(M)$ to D , that is $R(M, D) = D \cup U^+(M) - U^-(M)$.

As shown in [51, 52], this technique is sound and complete:

1. (*Soundness*) for every stable model M of $\Pi(IC)_D$, $U^-(M)$ and $U^+(M)$ define a repair for D .
2. (*Completeness*) for every database repair R for D there exists a stable model M of $\Pi(IC)_D$ such that $R = D \cup U^+(M) - U^-(M)$;

Example 4.7 Consider the set of integrity constraints IC consisting of the constraints ic_1 and ic_2 defined in Example 4.6. Assume that the database instance D contains the facts $P(a), P(b), S(a)$ and $Q(a)$. The derived extended disjunctive program contains the two rules which are shown in Example 4.6, that is $DP(IC) = \{dj(ic_1), dj(ic_2)\}$. These rules can now be rewritten in standard form by eliminating body disjunctions, obtaining the program $\Pi(IC) = st(DP(IC))$.

The computation of the program $\Pi(IC)_D$, derived from the union of $\Pi(IC)$ with the facts in D , gives six stable models. The table in Figure 4.1 shows each stable model M of $\Pi(IC)_D$ and the corresponding sets of update operation $U^-(M)$ and $U^+(M)$ defining a repair for D . For instance, the first row of the table states that a possible repair for D consists of deleting the two atoms $P(b)$ and $Q(a)$ from the database instance $D = \{P(a), P(b), S(a), Q(a)\}$, obtaining $R(M, D) = \{P(a), S(a)\}$.

□

Stable Model M	$U^-(M)$	$U^+(M)$
$D \cup \{\neg P'(b), \neg Q'(a)\}$	$P(b), Q(a)$	\emptyset
$D \cup \{\neg P'(b), T'(a)\}$	$P(b)$	$T(a)$
$D \cup \{\neg Q'(a), S'(b)\}$	$Q(a)$	$S(b)$
$D \cup \{T'(a), S'(b)\}$	\emptyset	$T(a), S(b)$
$D \cup \{\neg Q'(a), Q'(b), T'(b)\}$	$Q(a)$	$Q(b), T(b)$
$D \cup \{Q'(b), T'(a), T'(b)\}$	\emptyset	$Q(b), T(a), T(b)$

Fig. 4.1. Stable Models and corresponding Repairs

Referential Integrity Constraints

In the presence of existential quantified variables the rewriting of constraints is modified. The following example informally presents how referential integrity constraints are rewritten.

Example 4.8 Consider the referential constraint

$$\forall x [employee(x) \Rightarrow \exists y ssn(x, y)]$$

stating that every employee must have a social security number. Then the rewriting is as follows

$$\neg employee'(x) \vee ssn'(x, \perp) \leftarrow (employee(x) \vee employee'(x)), \\ (not\ ssn_{\pi}(x) \vee \neg ssn'_{\pi}(x))$$

where \perp denotes an unknown value and ssn_{π} and ssn'_{π} are new predicate symbols storing, respectively, the projection of ssn , and of ssn' , on the universal quantified variable x . They are defined by the rules

$$ssn_{\pi}(x) \leftarrow ssn(x, y) \\ ssn'_{\pi}(x) \leftarrow ssn'(x, y)$$

□

4.3.2 Computing Consistent Answers

We consider now the problem of computing a consistent answer without modifying the (possibly inconsistent) database.

A (relational) query over a database defines a function from the database to a relation. It can be expressed by means of alternative equivalent languages such as (i) relational algebra, (ii) safe, relational calculus or (iii) safe, non-recursive Datalog [1, 74] (i.e. safe Datalog without disjunction, classical negation and recursion). In this section we will use Datalog. Thus, a query is a pair (g, π) where π is a safe, non-recursive Datalog program and g is a predicate symbol specifying the output (derived) relation.

Let M be a stable model of $\Pi(IC)_D$ and $R(M, D)$ the repair obtained by means of the deletions and insertions “specified” by M . The answer to a query (g, π) over a repair $R(M, D)$ are the atoms $g(t)$ which are true in the stable models of the Datalog program $\pi \cup \Pi(IC) \cup R(M, D)$. In fact the program $\pi \cup \Pi(IC) \cup R(M, D)$ admits a unique stable minimal model consisting of the stable model of the program $\pi_{R(M, D)}$, i.e. of the program $\pi \cup R(M, D)$.

The computation of the consistent answers of a query (g, π) over the database D with integrity constraints IC can be derived by considering for each stable model M of $\Pi(IC)_D$ (which defines the repair $R(M, D)$), the answer to the program $\pi_{R(M, D)}$.

The consistent answer of the query $Q = (g, \pi)$ over the database D under constraints IC , denoted as $CQA(Q, D, IC)$, gives three sets, denoted $CQA(Q, D, IC)^+$, $CQA(Q, D, IC)^-$ and $CQA(Q, D, IC)^u$. These contain, respectively, the sets of g -tuples which are *true* (i.e. belonging to $Q(R)$ for all repairs R), *false* (i.e. not belonging to $Q(R)$ for all repairs R) and *undefined* (i.e. set of tuples which are neither true nor false).

Let us denote with $\mathcal{SM}(\Pi)$ be the set of stable models of the logic program Π . The consistent answers for the query $Q = (g, \pi)$ are as follows

$$\begin{aligned} CQA(Q, D, IC)^+ &= \{ g(t) \mid \forall M \in \mathcal{SM}(\Pi(IC)_D), g(t) \in \mathcal{SM}(\pi_{R(M, D)}) \} \\ CQA(Q, D, IC)^- &= \{ g(t) \mid \nexists M \in \mathcal{SM}(\Pi(IC)_D) \text{ s.t. } g(t) \in \mathcal{SM}(\pi_{R(M, D)}) \} \\ CQA(Q, D, IC)^u &= \{ g(t) \mid \exists M_1, M_2 \in \mathcal{SM}(\Pi(IC)_D) \text{ such that} \\ &\quad g(t) \in \mathcal{SM}(\pi_{R(M_1, D)}) \wedge g(t) \notin \mathcal{SM}(\pi_{R(M_2, D)}) \} \end{aligned}$$

Observe that if $Q = (g, \emptyset)$ for each model $M \in \mathcal{SM}(\Pi(IC)_D)$, we have that $\pi_{R(M, D)}$ is the repair $R(M, D)$.

Example 4.9 Consider the database D and the set of integrity constraints IC of Example 4.7. The program $\Pi(IC)_D$ has the stable models in Figure 4.1. The consistent answers the query $Q = (S, \emptyset)$ (with empty program) are as follows: $CQA(Q, D, IC)^+ = \{S(a)\}$, $CQA(Q, D, IC)^u = \{S(b)\}$ and $CQA(Q, D, IC)^-$ consists of the atoms $S(u)$ not belonging to $CQA(Q, D, IC)^+$ and $CQA(Q, D, IC)^u$. □

This technique is general, but expensive. Given a database D , a query $Q = (g, \pi)$ and a set of full, single-head integrity constraints IC , checking if some fact belongs to the consistent answer of Q is in Π_2^P .

In [51, 52] it has been shown that, if IC is either a set of functional dependencies or a set of full inclusion dependencies and $Q = (g, \emptyset)$ (with query program empty), then the consistent answer to Q can be computed in polynomial time. Observe that, these tractable cases have been implicitly identified also in [4] (cfr. Section 2.5) and subsequently in [29] (cfr. Section 3.2.1).

Moreover, given a set functional dependencies FD , a query $Q = (g, \pi)$ and a ground tuple $t = g(a_1, \dots, a_n)$, checking whether

1. $t \in CQA(Q, D, IC)^+$ is *coNP*-complete,

2. $t \in CQA(Q, D, IC)^-$ is *coNP*-complete,
3. $t \in CQA(Q, D, IC)^u$ is *NP*-complete.

Note that, the *coNP*-hardness for $CQA(Q, D, IC)^+$ has also been proved in [7, 8] (cfr. Section 3.1) and subsequently in [29] (cfr. Section 3.2.1).

4.3.3 Repair Constraints

In the integration of databases, the presence of inconsistent data may be resolved by repairing the integrated database. *Repair constraints* can be defined during the integration phase to give preference to certain data with respect to others and to define which repairs are feasible. Thus, the number of repairs can be restricted.

A *repair constraint* is a denial rule of the form

$$\leftarrow up_1(\alpha_1), \dots, up_m(\alpha_m), L_1, \dots, L_n$$

where $up_1, \dots, up_m \in \{insert, delete\}$, $\alpha_1, \dots, \alpha_m$ are (standard) atoms and L_1, \dots, L_n are (standard) literals.

The semantics of a repair constraint is as follows: if the conjunction L_1, \dots, L_n is true in the repaired database, then at least one of the update operations $up_i(\alpha_i)$ must be false.

Let D be a database, IC a set of integrity constraints and RC a set of repair constraints. A repair R for D satisfies RC if for each repair constraint

$$\leftarrow insert(\alpha_1), \dots, insert(\alpha_k), delete(\alpha_{k+1}), \dots, delete(\alpha_m), L_1, \dots, L_n$$

then

- i) there is some α_i with $1 \leq i \leq k$ which is not in $\{Facts(R) - Facts(D)\}$,
or
- ii) there is some α_i with $k+1 \leq i \leq m$ which is not in $\{Facts(D) - Facts(R)\}$,
or
- iii) there is some L_i false in $Facts(R)$.

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

Example 4.10 Consider the database instance D containing information about names and salaries of employees. The facts in D are the following

$$\{employee(Peter, 30000), employee(John, 40000), employee(John, 50000)\}$$

Given the integrity constraint

$$\forall(x, y, z)[employee(x, y), employee(x, z) \Rightarrow x = z]$$

There are two repairs for D , namely R_1 and R_2 with

$$Facts(R_1) = \{employee(Peter, 30000), employee(John, 40000)\}$$

$$Facts(R_2) = \{employee(Peter, 30000), employee(John, 50000)\}$$

The following repair constraint states that if the same employee occurs with more than one salary, then the tuple with the lowest salary cannot be deleted.

$$\leftarrow delete(employee(x, y)), employee(x, z), z > y$$

Thus, it makes R_2 not feasible since $employee(John, 40000)$ is deleted (i.e. it is in $\{Facts(D) - Facts(R_2)\}$ and R_2 contains $employee(John, 50000)$ with $50000 > 40000$.

□

The formal semantics of databases with both integrity and repair constraints is given by rewriting the repair constraints into extended rules with empty heads (denials). Specifically, the sets of integrity constraints IC and repair constraints RC are rewritten into an Extended Disjunctive Logic Program Π . Each stable model of Π over a database D can be used to generate a repair for the database, whereas each stable model of the program $\Pi \cup \pi$, over the database D , can be used to compute a consistent answer of a query (g, π) . Each model defines a set of actions (update operations) over the inconsistent database to achieve a consistent state.

Let r be a repair constraint of the form

$$\leftarrow insert(\alpha_1), \dots, insert(\alpha_k), delete(\alpha_{k+1}), \dots, delete(\alpha_m), \\ \beta_1, \dots, \beta_l, not \beta_{l+1}, \dots, not \beta_n, \varphi$$

where $\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_n$ are base atoms and φ is a conjunction of built-in atoms. Then $dj(r)$ denotes the denial rule

$$\leftarrow \alpha'_1, \dots, \alpha'_k, \neg \alpha'_{k+1}, \dots, \neg \alpha'_m, ((\beta_1, not \neg \beta'_1) \vee \beta'_1), \dots, ((\beta_l, not \neg \beta'_l) \vee \beta'_l), \\ ((not \beta_{l+1}, not \beta'_{l+1}) \vee \neg \beta'_{l+1}), \dots, ((not \beta_n, not \beta'_n) \vee \neg \beta'_n), \varphi$$

where γ'_i denotes the atom derived from γ by replacing the predicate symbol P of γ with the new symbol P' , that is, if γ is $P(u)$ then γ'_i is $P'(u)$.

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

- i) some atom α'_i ($1 \leq i \leq k$) must be *false*, i.e. α_i is not inserted in the database, or
- ii) some atom $\neg \alpha'_j$ ($k+1 \leq j \leq m$) must be *false*, i.e. α_j is not deleted from the database, or
- iii) some formula $((\beta_i, not \neg \beta'_i) \vee \beta'_i)$ ($1 \leq i \leq l$) must be *false*, i.e. the atom β_i is *false* in the repair, or
- iv) some formula $((not \beta_j, not \beta'_j) \vee \neg \beta'_j)$ ($l+1 \leq j \leq n$) must be *false*, i.e. the atom β_j is *false* in the repair, or
- v) the conjunction of built-in literals φ must be *false*.

Observe that, the formula $((\beta_i, not \neg \beta'_i) \vee \beta'_i)$ states that either β_i is in the source database D and it is not deleted from D or β_i is inserted into D .

Analogously, the formula $((\text{not } \beta_j, \text{not } \beta'_j) \vee \neg \beta'_j)$ states that either the atom β_j is *false* in the source database and is not inserted by the repair or it is deleted from the database by the repair.

Example 4.11 If we consider the following repair constraint

$$\leftarrow \text{insert}(P(a)), \text{delete}(P(b)), S(c), \text{not } Q(b)$$

then the derived denial rule is

$$\leftarrow P'(a), \neg P'(b), ((S(c), \text{not } \neg S'(c)) \vee S'(c)), ((\text{not } Q(b), \text{not } Q'(b)) \vee \neg Q'(b))$$

where $((S(c), \text{not } \neg S'(c)) \vee S'(c))$ means that either $S(c)$ is present in the source database and it is not derived as *false*, or it is derived as *true*. Analogously, $((\text{not } Q(b), \text{not } Q'(b)) \vee \neg Q'(b))$ means that either $Q(b)$ is not present in the database and it is not derived as *true* or it is derived as *false*. \square

Let RC be a set of integrity constraints, then $DP(RC) = \{ dj(rc) \mid rc \in IC \}$ and $\Pi(RC) = st(DP(RC))$. Thus, $DP(RC)$ denotes the set of (generalized) disjunctive rules derived from the rewriting of RC , whereas $\Pi(RC)$ denotes the set of (standard) disjunctive rules derived from $DP(RC)$ by rewriting body disjunctions.

Given a database D , a set of integrity constraints IC and a set of repair constraints RC , $\Pi(IC, RC)_D$ denotes the logic program derived from the union of the rules in $\Pi(IC) \cup \Pi(RC)$ with the facts in D .

It has been shown that the rewriting technique presented above is sound and complete: each stable model M of $\Pi(IC, RC)_D$ defines a *feasible* repair $R(M, D)$ (obtained interpreting the primed predicates, as shown in Section 4.3.1), and for each *feasible* repair R there is a stable model M of $\Pi(IC, RC)_D$ such that R can be obtained by interpreting M .

4.3.4 Prioritized Repairs

A *prioritized update* is a rule which gives the possibility of expressing preference among update operations and, consequently, to give preference to some repairs for the database with respect to others. Integrity constraints with prioritized updates can be rewritten into prioritized extended disjunctive rules [73].

A *prioritized update rule* is of the form

$$up_1(\alpha) \preceq up_2(\beta)$$

where $up_1, up_2 \in \{\text{insert}, \text{delete}\}$ and α and β are atoms.

Given a set of prioritized updates PC , we will denote as PC^* the reflexive, transitive closure of PC . A set of prioritized updated PC is said to be consistent if there are not two prioritized updated rules in PC^* of the form

$up_1(\alpha') \preceq up_2(\beta')$ and $up_1(\beta'') \preceq up_2(\alpha'')$ such that α' unifies with α'' and β' unifies with β'' .

Given a repair R for a database D , we denote as $update(R)$ the update atoms derived from R , that is

$$update(R) = \{insert(\alpha) \mid \alpha \in \{Facts(R) - Facts(D)\}\} \cup \{delete(\alpha) \mid \alpha \in \{Facts(D) - Facts(R)\}\}$$

Given a database D , a set of integrity constraints IC , a set of repair constraints RC , the relation \sqsubseteq is defined over the repairs for D w.r.t. IC and RC . For any repair R_1, R_2 and R_3 for D ,

1. $R_1 \sqsubseteq R_1$;
2. $R_1 \sqsubseteq R_2$ if
 - a) $\exists u_1 \in \{update(R_1) - update(R_2)\}, \exists u_2 \in \{update(R_2) - update(R_1)\}$ such that $u_1 \preceq u_2 \in PC^*$ and
 - b) $\nexists u_3 \in \{update(R_1) - update(R_2)\}$ such that $u_2 \preceq u_3 \in PC^*$;
3. 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 . A repair R for D is said to be *preferred* if there is no repair R' for D such that $R' \neq R$ and $R \sqsubseteq R'$.

Given a database D , a set of integrity constraints IC , a set of repair constraints RC and a set of prioritized constraints PC , a Prioritized Disjunctive Logic Program $(\Pi(IC, RC)_D, \Phi(PC))$ is obtained adding the Disjunctive Logic Program $\Pi(IC, RC)_D$, obtained by rewriting of $IC \cup RC$ into disjunctive rules, and the prioritized rules $\Phi(PC)$, obtained from rewriting of PC .

Also in this case, the rewriting technique is sound and complete, that is each preferred stable model of $(\Pi(IC, RC)_D, \Phi(PC))$ defines a preferred repair for D and each preferred repair is derived from a preferred stable model.

Observe that, it is possible to consider more complex prioritized rules, as that of the following example, since they can be reduced to the basic case [73].

Example 4.12 Consider the database and the integrity constraint of Example 4.10. Assume that the following prioritized update is defined

$$delete(employee(x, y)) \preceq delete(employee(x, z)) \leftarrow y < z$$

stating that, between two distinct instances of the same employee, we prefer to delete the one with higher salary. As shown in Example 4.10 there are two repairs for the database: the former R_1 consists of deleting the fact $employee(John, 50000)$ from D , whereas the latter consists of deleting $employee(John, 40000)$ from D . Thus, R_1 is preferable to R_2 . □

The introduction of prioritized update rules increases the complexity of computing repairs and answers from the second to the third level of the polynomial hierarchy, since the complexity of checking if a given atom belongs to some or all preferred stable models is complete for the third level of the polynomial hierarchy.

4.4 Discussion

In this chapter we have presented some techniques that exploits the expressive power of (*extended*) *disjunctive logic programs* (with both *classical* negation and *negation as failure*) for specifying repairs for a (possible inconsistent) database. Specifically, in [5, 9] (cfr. Section 4.2) repairs have been specified by means of a *disjunctive logic program with exceptions* with e-answer set semantics. This logic program can be transformed into an extended disjunctive logic program (without exceptions) with answer set semantics. As shown in [50], the resulting program can be further transformed into an equivalent disjunctive normal logic program with stable model semantics [49]. In [51, 52] (cfr. Section 4.3) repairs have been specified by an *extended disjunctive logic program* derived from the set of integrity constraints. Moreover, *repair constraints* and *prioritized update rules* have been introduced. The former ones allow us to specify conditions under which repairs are feasible, and both (classical) constraints and repair constraints can be still rewritten into an extended disjunctive logic program which specifies repairs for a database. Prioritized updates are rules which allow us to give preference to some repairs for the database with respect to others. With the introduction of prioritized updates the rewriting yields a set of *prioritized extended disjunctive rules* [73].

The two approaches above are very similar. Although the former has been formalized in presence universal binary constraints, it can be extended to work with (general) universal constraints in standard format (cfr. Section 1.3.1). But, when an universal constraint is rewritten, the number of *disjunctive stabilizing rules* (cfr. Section 4.2.2) grows according to the number of subsets of the database literals in the constraint, i.e. exponentially in the size of the constraint. Similarly, in the latter approach, when body disjunction is eliminated from a rule (which is obtained by rewriting a constraint), the body-disjunction rules generated are exponential in the number the body literals. Moreover, both the techniques deal with referential integrity constraints in a very similar manner (cfr. Section 4.2.4 and Section 4.3.1).

Different extensions of these techniques have been proposed. In [51, 52] *repair constraints* and *prioritized repairs* have been introduced (cfr. Section 4.3.3 and Section 4.3.4, respectively). Whereas in [8] an alternative semantics for repairs based on the *minimum number* of changes, instead of the *minimal set* of changes, has been presented. In this case the repair program is obtained replacing the persistence defaults by *weak constraints* [21], which are not imposed on the original database, but on the answer sets of the program. This repair semantics was further studied in [65, 66].

The approaches discussed above are based on classical logic, and database repairs correspond to certain minimal models of the program that specifies repairs. In [6, 11] repairs are specified in a non-classical logic: the *Annotated Predicate Calculus* [56]. In this case repairs correspond to some distinguished minimal models of a theory written in annotated predicate logic. The approach of [6] is applicable to queries that are conjunctions or disjunctions

of positive literals and to universal constraints. The specification methodology was extended from universal integrity constraints to referential integrity constraints in [11]. In [12] repairs have been specified by means of disjunctive logic programs with stable model semantics. The database predicates in these programs contain *annotations* as extra arguments (as apposed to annotated programs that contain annotated atoms). The approach works for first-order queries in presence of universal constraints and referential integrity constraints. Moreover, the use of annotations as extra argument in predicates entails that the number of rules generated by rewriting an integrity constraint is linear in the number of literals of the constraint.

The Attribute-Based Repairing Paradigm

The logical characterization of consistent query answers was provided on the basis of the notion of repair for a database [4]. A repair is defined as a consistent database instance that minimally differs from the original instance, according to some sort of *distance* between the original instance and the repaired one. We point out that a property shared by the notions of repair presented in Chapter 3 and Chapter 4 is that they are defined by inserting and/or deleting whole tuples. Moreover, the concept of repair is exploited for the characterization of consistent answers, but the problem of explicitly computing a repair has not received much attention in the techniques surveyed in Chapter 3 and Chapter 4. This has been often motivated by the consideration that may be an exponential number of repairs for an inconsistent database.

On the other hand, in several scenarios it is interesting computing a repair which is *minimal* according to a *numerical distance* from the original database instance. In these scenarios repairs are obtained by performing value updated, i.e. working at attribute-level, rather than at tuple-level. Thus, the distance is a numerical function over the attribute-values which are updated in order to restore the consistency. In this chapter we present several works which investigate the problem of computing a repair for a database working at attribute-level, i.e. adopting the *attribute-based repairing paradigm*.

Example 5.1 Suppose that we have the database consisting of the relation *Dioxin*(*Sample*, *SampleDate*, *Food*, *AnalysisDate*, *Lab*, *DioxinLevel*), whose instance is shown in Figure 5.1. The relation reports the dioxin levels in food samples. Assume that it is defined the following constraints, which imposes that the date of analyzing a given sample cannot precede the date the sample was taken.

$$\forall x_1, x_2, x_3, x_4, x_5, x_6 [Dioxin(x_1, x_2, x_3, x_4, x_5, x_6) \Rightarrow x_2 \leq x_4]$$

The first tuple in the Dioxin Database says that the sample ‘110’ was taken on ‘17 Jan 2002’ and analyzed the day after at the ‘*ICT*’ lab, and that

<i>Sample</i>	<i>SampleDate</i>	<i>Food</i>	<i>AnalysisDate</i>	<i>Lab</i>	<i>DioxinLevel</i>
110	17 Jan 2002	poultry	18 Jan 2002	ICI	normal
220	17 Jan 2002	poultry	16 Jan 2002	ICB	alarming
330	18 Jan 2002	beef	18 Jan 2002	ICB	normal

Fig. 5.1. Dioxin Database

the dioxin level of this sample was normal. While sample ‘110’ respects the constraint, sample ‘220’ violates it.

□

Following the *tuple-based* repairing paradigm there is a unique repair for the database above, which is obtained by simply deleting the faulty tuple about sample ‘220’. The query “*Get the alarming samples*” will then yield the empty consistent answer. Nevertheless, it seems more reasonable to conclude that there was a sample, whose number is ‘220’, with an alarming dioxin level, but either the sample or the analysis date contains an error. The problem with tuple-based repairing paradigm is that an entire tuple may be deleted, even if only a minor part of it is erroneous. In order to avoid deleting the entire tuple, finer repair primitives that enable correcting faulty values within a tuple are necessary. We will say that approaches capable of repairing at the attribute-value level follow the *attribute-based repairing paradigm*.

In the Dioxin database the inconsistency can be “cleaned” in several ways, for example, by antedating the sample date or by postdating the analysis date of the sample ‘220’.

5.1 Repairing Census Data

In [42] a framework for repairing inconsistent databases considering the specific domain of census data has been proposed. A databases consisting of one fixed relation scheme has been used for encode data contained in a census questionnaire. Thus, the problem addressed in [42] is that of finding a repair of a (possible inconsistent) relation instance with respect to the set of constraints (expressed as first order formulas) that every questionnaire have to satisfy. Disjunctive logic programming extended with weak and strong constraints has been used for computing repairs for the database encoding the census data.

The notion of minimality exploited for computing repairs is based on the number of attribute-value changed followed by a preference criterion. First, minimal repairs are selected according that they are the consistent databases in which the least number of attribute values w.r.t. the original database is changed. Then, a preferred repair is produced by choosing it among the minimal ones. The preferred models of the logic program encoding the problem

correspond to the preferred repairs of the relation associated with a census questionnaire.

5.1.1 Repairs for Census Data

Census data is collected by means of questionnaires, each one including the details of the persons living together in the same house. Each questionnaire is focused around the notion of *reference person*. All persons living in private households are identified from their *relationship* to the reference member of the household. A basic questionnaire can be represented by instances of the following relation scheme:

$$Qst(PersonId, Relationship, Sex, Age, MaritalStatus)$$

where *PersonId* is an integer number used to identify members of the household (it is the key of *Qst*), with the number “1” reserved to identify the reference person. Each person living in a private household is classified with respect to the reference person by means of the attribute *Relationship*. Moreover, the *structural* attributes *Sex*, *Age* and *MaritalStatus* are considered for each member of the household. Observe that, each questionnaire refers to only one household.

Each questionnaire must satisfy a set of *edit rules* used for validating the collected data. The edit rules, encoded through first order formulas, play the role of integrity constraints *IC*, over the relation *Qst*.

The *Qst* relation may contains *null* values, which encode missing answer in the questionnaire. We say that an instance *I* of *Qst* is consistent if no *null* value occur in its tuple and $I \models IC$, inconsistent otherwise.

Example 5.2 The following relation instance encodes the data of an household with two person which are married (the reference person is the woman).

<i>PersonId</i>	<i>Relationship</i>	<i>Sex</i>	<i>Age</i>	<i>MaritalStatus</i>
1	<i>reference</i>	<i>F</i>	31	<i>married</i>
2	<i>spouse</i>	<i>M</i>	49	<i>married</i>

Fig. 5.2. A instance of the *Qst* relation

Some edit rule that must be satisfied are the following:

- a) a *spouse* of the reference person should be *married* (but not necessarily vice versa):

$$\forall x \exists y, z_1, z_2 \quad Qst(x, spouse, z_1, z_2, y) \Rightarrow y = married$$

- b) cohabitant *partners* of the reference person can not be married with somebody else (i.e., they should be either *single*, or *divorced*, or *widowed*):

$$\forall x \exists y, z_1, z_2 \ Qst(x, partner, z_1, z_2, y) \Rightarrow y \neq married$$

- c) the reference person cannot be married with more than one person:

$$\forall x, y \exists z_1, z_2, z_3, z_4, z_5, z_6 \ Qst(x, spouse, z_1, z_2, z_3) \wedge \\ Qst(y, spouse, z_4, z_5, z_6) \Rightarrow x = y$$

Clearly, the questionnaire in Figure 5.2 is consistent w.r.t. the set of constraints $IC = \{a, b, c\}$.

□

We now introduce the notion of repair for inconsistent questionnaires.

For two tuples $t = \langle v_1, \dots, v_m \rangle$ and $t' = \langle v'_1, \dots, v'_m \rangle$, let $dist(t, t')$ be the number of values v_j occurring in t that differ from the corresponding values in t' , i.e., such that $v_j \neq v'_j$. Further, let ρ be a mapping from tuples over Qst to tuples over Qst .

Given an instance I of Qst that is inconsistent w.r.t. the set of integrity constraints IC , a (minimal) repair R for I w.r.t. IC is a relation $R = \{\rho(t) \mid t \in I\}$ such that

- i) R is consistent w.r.t. IC ($R \models IC$), and
- ii) $\sum_{t \in I} dist(t, \rho(t))$ is minimum over the set of relations which are consistent w.r.t. IC , i.e., the number of attribute-values changed between the tuple in I and that in R is the minimum number of changes over all possible consistent relations.

Example 5.3 Consider the following inconsistent relation w.r.t. the set of constraints $IC = \{a, b, c\}$ of the Example 5.2.

<i>PersonId</i>	<i>Relationship</i>	<i>Sex</i>	<i>Age</i>	<i>MaritalStatus</i>
1	<i>reference</i>	<i>F</i>	31	<i>married</i>
2	<i>spouse</i>	<i>M</i>	49	<i>null</i>

Fig. 5.3. A inconsistent instance of the Qst relation

A (minimal) repair for the relation in Figure 5.3 is the relation in Figure 5.2. The distance between the two relations is one, i.e. the cardinality of the set of values changed. Observe that the (consistent) relation consisting of the first tuple unaltered and the second tuple changed in $\langle 2, partner, M, single \rangle$ is not a (minimal) repair because of the distance between this relation and the original one is not minimal.

□

In general, there are many possible minimal repairs for an inconsistent questionnaire. The objective is that of choosing one of the minimal repairs without altering the statistical properties of the census. This can be obtained by exploiting some extra information about the census domain, possibly old statistics concerning the same population. Such information can be encoded through a set of first order formulas called *preference rules*, used for expressing some preferences over the repairs R for Qst . Intuitively, a preference rule specifies a preferred way to repair the data of a person under some circumstances. The preference rules should be satisfied by as many persons in the household as possible.

Example 5.4 The following preference rule expresses that it is likely for a married person living in the household, whose relationship with the reference person is unknown, to be his/her spouse.

$$\forall x_1, y_1, x_2, y_2 \quad Qst(1, reference, x_1, y_1, married) \wedge \\ Qst(x, null, x_2, y_2, married) \Rightarrow R(x, spouse, x_2, y_2, married)$$

Observe that the predicate symbol R refers to the repair of Qst . □

A *preferred repair* R_P for a questionnaire Qst is a minimal repair such that the number of satisfied preference rules is the greatest over all minimal repairs for Qst .

Thus, given a questionnaire Qst , a set of integrity constraints IC and a set of preference rules Φ , the problem addressed is that of finding a preferred (minimal) repair for Qst .

Example 5.5 Consider the relation in Figure 5.4, which is inconsistent w.r.t. the set of constraints $IC = \{a, b, c\}$ of the Example 5.2. Assume that the preference rule of Example 5.4 is specified.

<i>PersonId</i>	<i>Relationship</i>	<i>Sex</i>	<i>Age</i>	<i>MaritalStatus</i>
1	<i>reference</i>	<i>F</i>	31	<i>married</i>
2	<i>null</i>	<i>M</i>	49	<i>married</i>

Fig. 5.4. A inconsistent instance of the Qst relation

Note that there is no unique way to provide the missing ‘relationship’ value for the second tuple. For instance, the person identified by ‘2’ may be the father or the spouse of the reference person (but not a partner, otherwise the constraint b) would be violated). These alternative belong to the set of the minimal repairs for the relation.

However, as specified by the preference rule stated above, most married couples live together, and hence we should probably prefer the repair in which the second tuple is $\langle 2, spouse, M, married \rangle$. In fact, the preferred repair is the relation shown in Figure 5.2. \square

5.1.2 Computing Repairs

The problem of finding a preferred repair for a questionnaire has been solved using a *disjunctive logic program with weak constraints* [21] whose preferred models (namely, *best models*) correspond to the preferred (minimal) repairs of a given questionnaire.

A disjunctive logic programming language with constraints (DLP^w) includes rules, *weak* and *strong* constraints. The presence of *strong* constraints modifies the semantics of the program by discarding all models which do not satisfy some of them. Whereas, the semantics of *weak* constraints minimizes the number of violated instances of constraints. Weak constraints are very powerful for capturing the concept of ‘preference’ in commonsense reasoning. Preferences may have different priorities, therefore weak constraints in DLP^w can be assigned different priorities as well, according to their ‘importance’. A weak constraint is defined *stronger than* another weak constraint if it has a higher priority than the other. Informally, the semantics of a DLP^w program Π is given by the stable models of the set of the rules of Π satisfying all strong constraints and minimizing the number of violated weak constraints according to the prioritization.

A preferred (minimal) repair R for a questionnaire encoded in an instance of Qst w.r.t. a set of integrity constraints IC and a set of preference rules Φ is a relation such that (i) it is consistent w.r.t. IC ; (ii) the number of values changed is minimized; and (iii) the number of satisfied preference rules in Φ is maximizes.

Observe that the constraints in IC are *strong* in the sense that in order to obtain a consistent relation (not necessarily a minimal repair) they must be satisfied. Therefore, they are translated into strong constraints of the disjunctive logic program used for computing preferred repairs. On the other hand, in order to obtain a minimal repair the number of values changed must be minimized. Thus, weak constraints can be used assigning a penalty to each value change made to the input questionnaire. Finally, to maximize the number of satisfied preference rules again weak constraints are used. Specifically, a weak constraint for each preference rule in Φ is created. This constraint assigns a penalty for each violation of the corresponding preference rule. These weak constraints have a lower priority than the weak constraints enforcing minimizations of changes because of the preferred repairs are chosen among the minimal repairs for the questionnaire.

The system DLV [37], which supports the disjunctive logic programming language DLP^w , has been used for obtaining an implementation of the method presented for repairing census data.

5.2 Complexity and Approximation of Repairing Numerical Data

In [16] the problem of repairing databases by changing integer numerical values at the attribute level, with respect to denial constraints, has been studied. In this context, the authors introduce a quantitative definition of database repair, based on the square of the Euclidean distance between the originals and modified values in the database instance. In this case, the ‘minimal’ repairs are that minimizing the quantitative global distance, changing only some *fixable* attribute values and keeping the values for the attributes in the key of the relations.

The *database fix problem*, namely the problem of determining the existence of a repair at a distance not bigger than a given bound, has been addressed. In particular, the problems of construction and verification of such a repair have been studied. Moreover, the problem of deciding the consistency of query answers has been studied.

First, undecidability for the problem of checking the existence of minimal repairs has been proved in the presence of (non-linear multi-attribute) aggregate constraints. Then, the authors concentrate on denial constraints, proving NP -completeness for the database fix problem under a subclass of denial constraints, namely the *linear* denial constraints. Moreover, it has been proved that the problem of finding the minimum distance from a database to a minimal repair is $MAXSNP$ -hard in general. Thus, for a subclass of denial constraints (called *local* denial constraints) an approximation within a constant factor has been provided.

5.2.1 Least Square Repairs

For each relation, it is assumed that there is a set of all the *fixable* attributes, that takes values in \mathbb{Z} and are allowed to be fixed. We will denote as \mathcal{F} the set of all fixable attribute. Attributes outside \mathcal{F} are called *rigid* (clearly, we may also have rigid numerical attributes). Moreover, for each relation, the primary key is defined on rigid attributes. In the following we assume that the key constraints are satisfied both by the initial database instance and its repairs (also called *fixes* in this context).

A *linear denial constraint* has the form $\forall X \neg(P_1 \wedge \dots \wedge P_n)$, where the P_i are database atoms, or built-in atoms of the form $x\theta c$, where x is a variable, c is a constant and $\theta \in \{=, \neq, <, >, \leq, \geq\}$, or $x = y$. If $x \neq y$ is allowed, we call them *extended linear denials*. Observe that, with either extended or linear denial constraints we can not use built-in atoms of the form $x \leq y$.

An *aggregation constraint* is given by the aggregate operators *sum*, *count*, *distinct* and *average*. *Filtering* aggregation constraints impose conditions on the tuples over which aggregation is applied, e.g. $sum(A_1 : A_2 = 3) > 5$ is a sum over A_1 of tuples such that $A_2 = 3$. *Multi-attribute* aggregation constraints allow arithmetical combinations of attributes as arguments for *sum*, e.g. $sum(A_1 + A_2) > 5$ and $sum(A_1 \times A_2) > 100$. It is assumed that the aggregation constraints have attributes from only one relation.

We now introduce the *square distance* between two database, which is used for determining partial order among repairs for a given databases.

Since the original database instance and a repair share the same key values, we can use them to compute variations in the numerical values. We will denote as $t(k, P, D)$ the unique tuple t in the relation P in the instance D whose key value is k . To each fixable attribute $A \in \mathcal{F}$ a fixed numerical weight α_A is assigned. Let D and D' be two database instance over the same scheme with the set \mathcal{P} of relation names. Let $val(K_P)$ be the set of key values for a relation $P \in \mathcal{P}$, which is shared by D and D' .

The *square distance* between D and D' is

$$\Delta_{\bar{\alpha}}(D, D') = \sum_{P \in \mathcal{P}, A \in \mathcal{F}, k \in val(K_P)} \alpha_A [\pi_A(t(k, P, D)) - \pi_A(t(k, P, D'))]^2$$

where π_A is the projection on attribute A and $\bar{\alpha} = (\alpha_A)_{A \in \mathcal{F}}$.

Let D be a database instance D , \mathcal{F} a set of fixable attributes and IC a set of integrity constraints. Given a set of primary key dependencies KD such that $D \models KD$, a *repair* (or *fix*) for D w.r.t. IC is a database instance R such that:

- (a) R has the same scheme and domain as D ;
- (b) R has the same values as D for each attribute $A \notin \mathcal{F}$;
- (c) $R \models KD$;
- (d) $R \models IC$.

A *least squares repair* (LS-repair) for D is a repair R that minimizes the square distance $\Delta_{\bar{\alpha}}(D, R)$ over all the instances that satisfy (a)-(d).

Example 5.6 Consider the database instance D having the relations *Client*(*IDclient*, *Age*, *Amount*) and *Buy*(*IDclient*, *Item*, *Price*) with primary keys $\{IDclient\}$ and $\{IDclient, Item\}$, respectively. The set of constrains defined consists of the following *linear* denials

$$\forall x_1, x_2, x_3, x_4, x_5 \neg[Buy(x_1, x_2, x_3) \wedge Client(x_1, x_4, x_5) \wedge x_4 < 18 \wedge x_3 > 25]$$

$$\forall x_1, x_2, x_3 \neg[Client(x_1, x_2, x_3) \wedge x_2 < 18 \wedge x_3 > 50]$$

They require that people younger than 18 cannot spend more than 25 on one item, nor spend more than 50 in the store.

Assume that the database instance is that in Figure 5.5. The former denial constraint is violated by the tuples $\{t_1, t_4\}$ and $\{t_1, t_5\}$, whereas the latter by $\{t_1\}$ and $\{t_2\}$.

	<i>IDclient</i>	<i>Age</i>	<i>Amount</i>
t_1	1	15	52
t_2	2	16	51
t_3	3	60	900

	<i>IDclient</i>	<i>Item</i>	<i>Price</i>
t_4	1	<i>CD</i>	27
t_5	1	<i>DVD</i>	26
t_6	3	<i>DVD</i>	40

Fig. 5.5. Relations *Client* and *Buy*

The set of fixable attributes is $\mathcal{F} = \{Age, Amount, Price\}$ and $\alpha_A = 1$ for each $A \in \mathcal{F}$, there are two least square repairs, which are shown in Figure 5.6 and 5.7, respectively (t'_i is the modified version of the tuple t_i).

	<i>IDclient</i>	<i>Age</i>	<i>Amount</i>
t'_1	1	15	50
t'_2	2	16	50
t_3	3	60	900

	<i>IDclient</i>	<i>Item</i>	<i>Price</i>
t'_4	1	<i>CD</i>	25
t'_5	1	<i>DVD</i>	25
t_6	3	<i>DVD</i>	40

Fig. 5.6. Repair R_1 for the relations *Client* and *Buy*

	<i>IDclient</i>	<i>Age</i>	<i>Amount</i>
t'_1	1	18	52
t'_2	2	16	50
t_3	3	60	900

	<i>IDclient</i>	<i>Item</i>	<i>Price</i>
t_4	1	<i>CD</i>	27
t_5	1	<i>DVD</i>	26
t_6	3	<i>DVD</i>	40

Fig. 5.7. Repair R_2 for the relations *Client* and *Buy*

The distance of the LS-repair R_1 is $\Delta(D, R_1) = (52 - 50)^2 + (51 - 50)^2 + (27 - 25)^2 + (26 - 25)^2 = 10$ (since it is assumed $\alpha_A = 1$ for each $A \in \mathcal{F}$, we omits $\bar{\alpha}$ for simplicity). Similarly, the distance of the LS-repair R_2 is $\Delta(D, R_2) = (18 - 15)^2 + (51 - 50)^2 = 10$. □

5.2.2 Complexity Results and Approximations

In [16] it has been shown that under extended linear denials and filtering, multi-attribute aggregation constraints, the problems of existence of LS-repairs, and the consistent query answer problem are undecidable. This result is owing to the presence of filtering, multi-attribute aggregation constraints.

Indeed, considering only a (fixed) set of linear denial constraints, for a database instance D , checking the existence of an LS-repair R for D such

that $\Delta_{\bar{\alpha}}(D, R) \leq k$, with k a positive integer, is NP -complete. Moreover, for a fixed set of extended linear denial constraints: (i) The problem of checking if an instance R is an LS-repair for a database D is $coNP$ -complete, and (ii) the consistent query answer problem is in Π_2^P , and, for ground atomic queries it is Δ_2^P -hard.

It has been shown that the optimization problem of finding the minimum distance from an LS-repair R , w.r.t. a set of linear denial constraints, to a given input database instance D is a $MAXSNP$ -hard problem. Thus, unless $P = NP$, there is no *polynomial time approximation scheme* for this optimization problem [68]. After this negative result, the authors focus on finding an efficient algorithm for approximation within a constant factor. This has been done for a restricted class of linear denial constraints, namely the *local* linear denial constraints.

Indeed, for a fixed set of local denial constraints, the problem of finding the minimum distance from an LS-repair R to a database instance D can be solved by transforming this problem into an instance of the *Minimum Weighted Set Cover Optimization Problem* (MWSCP). This problem is $MAXSNP$ -hard, and its general approximation algorithms are within a logarithmic factor. By concentrating on local denials, a version of the MWSCP that can be approximated within a constant factor has been provided.

A set DC of linear denial constraints is said to be *local* if:

- i) attributes participating in equality atoms between attributes or in joins are all *rigid*;
- ii) there is a built-in atom with a *fixable* attribute in each element of DC ;
- iii) no attribute A appears in DC in both comparisons of the form $A < c_1$ and $A > c_2$ (in order to check this condition, the built atoms $x \leq c$, $x \geq c$, $x \neq y$ have to be expressed using only $<$ and $>$, e.g. $x \leq c$ is written as $x < c + 1$).

Local (linear denial) constraints have the property that by doing local fixes, no new inconsistencies are generated, and there is always an LS-repair with respect to them (a property that is not valid for (general) linear denial constraints).

As for (general) linear denial constraints, the problem of checking the existence of an LS-repair for a database within a constant distance, under local constraints, remains NP -complete. Further, the optimization problem of finding an LS-repair with the minimum distance from the original database is $MAXSNP$ -hard.

On the other hand, under local constraints, for a database instance D , the approximation algorithm based on MWSCP returns a repair \tilde{R} such that $\Delta_{\bar{\alpha}}(D, \tilde{R}) \leq c \times \Delta_{\bar{\alpha}}(D, R)$, where R is any LS-repair for D and c is a constant that depends on the number of atom in the set of local constraints.

One Database Atom Denials

Now we consider *one database atom* denial constraints, that is denial constraints which are restricted to have the form $\forall X \neg[P, B]$, where P is a predicate corresponding to a relation symbol, and B is a conjunction of built-in atoms. For this class of constraints, tractable cases for computing consistent answers under LS-repairs has been identified. This has been accomplished by reduction to computing consistent answers for (tuple-based and set-theoretic) repairs of the form introduced in [45] in presence of one key constraint.

The results and the algorithm introduced in [45] for computing consistent answers w.r.t. key constraints has been obtained for the class of conjunctive queries in \mathcal{C}_{tree} (cfr. Section 3.3.1). In [16] the authors exploits this results in order to show that, for one database atom denial constraints and queries in \mathcal{C}_{tree} , the consistent query answer under LS-repair is in *PTIME*.

Aggregate Conjunctive Queries

An *aggregate conjunctive query* has the form

$$Q(x_1, \dots, x_n; agg(z)) \leftarrow B(x_1, \dots, x_m, z, y_1, \dots, y_n)$$

where *agg* is an aggregation function (*sum*, *count distinct*, and *average*) and its *non-aggregate matrix* given by $Q'(x_1, \dots, x_n) \leftarrow B(x_1, \dots, x_m, z, y_1, \dots, y_n)$ is a first-order conjunctive query with built-in atoms, such that the aggregation attribute z does not appear among the x_i . An aggregate conjunctive query is cyclic (resp. acyclic) if its non-aggregate matrix is cyclic (resp. acyclic).

Example 5.7 The query $Q(x, y, sum(z)) \leftarrow P(x, y) \wedge Q(y, z, w) \wedge w \neq 3$ is an aggregate conjunctive query with aggregation attribute z . The corresponding non-aggregate matrix is $Q'(x, y) \leftarrow P(x, y) \wedge Q(y, z, w) \wedge w \neq 3$. Each answer $\langle x, y \rangle$ to Q' is expanded to $\langle x, y, sum(z) \rangle$ as answer to Q , where $sum(z)$ is the sum of all the values for z such that the body of the query is satisfied. \square

The semantics adopted for consistent query answers to aggregate queries is that firstly proposed in [7], namely the *range semantics*, which is based on the minimal range containing every values that the aggregation query can take in all possible repairs of the database. The consistent query answer decision problems under range semantics consist in determining if a query has its answer contained a range in every repair (cfr. Section 3.1).

In [16] it has been shown that, there is a fixed set of one database atom denial constraints and a fixed *aggregate acyclic conjunctive query*, such that the consistent query answer problem under the range semantics is *NP-hard*. Moreover, for any set of one database atom denials and conjunctive query with *sum* over a nonnegative attribute, a polynomial time approximation algorithm with a constant factor has been provided for deciding consistent answers under

range semantics. This approximation is obtained by reduction from *Bounded Degree Independent Set* which has efficient approximations within a constant factor that depends on the (bounded) degree.

5.3 An Heuristic for Repairing Inconsistent Databases

In [17] a *cost model* for computing database repairs as set of value modifications has been introduced. The authors observed a strong connection between the database repairing area and the *record linkage* field, also known as “duplicate removal” or “merge-purge”, which refers to the task of linking pairs of records that refer to the same entity in different data sets. There is a parallelism between searching for a repair for an inconsistent database instance and the record linkage task. Thus, it may be more helpful to automatically propose a repair in this situation.

The proposed cost model for repairs is based on two factors, *accuracy* and *similarity*. The *accuracy* of data is reflected in a weight $w(t)$ for each tuple t and represents the confidence placed by the user in the values therein. For example, tuples from a source S_1 may have weight greater than tuples from a source S_2 , reflecting a different degree of confidence in their accuracy. A variety of measure of *similarity* of data is available at the attribute or tuple level, for example, the string-edit distance.

The intractability of the problem of computing a minimum-cost repair has been shown. In light of this result the authors proposed an approach for repair construction based on on *equivalence classes* of pairs $(tuple, attribute)$ that are assigned identical values in minimum-cost repairs.

5.3.1 Minimum-Cost Repairs

Given a set of integrity constraints IC and a database instance D , a *repair* for D w.r.t. IC is a database instance R such that (i) R is consistent w.r.t. IC , and (ii) R contains the tuples in D possibly with modified attribute values, plus zero or more inserted tuples. Observe that, when attribute values are modified it may be the case that two from two distinct (conflicting) tuples we obtain one (consistent) tuple. For instance, given a relation scheme $P(A, B)$ where the primary key is A , if the instance consist of tuples $t_1 = \langle a_1, b_1 \rangle$ and $t_2 = \langle a_1, b_2 \rangle$, then a repair for P is the instance $\langle a_1, b_1 \rangle$.

A repair R for a database D has associated a *cost*, which is the sum of the costs of the actions (value modifications and tuple insertions) performed on D to obtain R .

Given a database instance D , we associate a *weight* $w(t) \geq 0$ to each tuple t in D . The cost of an attribute-level modification of a tuple t in a repair R is the weight $w(t) \geq 0$ times the distance between the original value of the attribute and its value in the repair. We assume that for two values v_1 and v_2 from the same domain, a *distance* function $dist(v_1, v_2)$ is available, with

lower values indicating greater similarity, according to a *similarity* metric. Moreover, it is assumed that a cost $insCost(P) > 0$ is associated with each relation P . It is the cost of inserting a tuple into P in a repair R .

In the following we will denote as $D(t, A)$ the value of a given attribute A of a tuple $t \in D$, and as $R(t, A)$ the value of the same tuple t in the repair R (it is assumed that during the repair process we can keep track of the tuple t). Further, for a set V of attribute, we use $D(t, V)$ (or $R(t, V)$) to represent the projection of t on attributes in V .

Thus, the *cost* relative to a tuple t in the repair R is the following:

$$cost(t) = \begin{cases} insCost(P), & \text{if } t \in new(P) \\ w(t) \cdot \sum_{A \in attr(P)} dist(D(t, A), R(t, A)), & \text{otherwise} \end{cases}$$

where $new(P)$ denotes the set of tuples inserted into P by R , and $attr(P)$ denotes the attribute of the relation P .

Example 5.8 Consider the database consisting of the relations *Customer*(*PhoneNumber*, *Name*, *Street*, *City*, *State*, *Zip*), and *Equipment*(*SerialNumber*, *PhoneNumber*, *Manufacturer*, *Model*). The *Customer* relation contains address information on customers, while the relation *Equipment* catalogs equipments installed at the customer location and includes manufacturer, model number and the serial number.

Assume that the set of constraints consists of the inclusion dependency $Equipment[PhoneNumber] \subseteq Customer[PhoneNumber]$, stating that each piece of equipment is associated with a valid customer, and the functional dependencies

- (i) $PhoneNumber \rightarrow Name, Street, City, State, Zip$;
- (ii) $Name, Street, Zip \rightarrow PhoneNumber$;
- (iii) $Zip \rightarrow City, State$; defined on *Customer* and
- (iv) $SerialNumber \rightarrow PhoneNumber, Manufacturer, Model$; defined on *Equipment*.

Consider the relation instances in Figure 5.8, where identifiers of tuples and the weight column $w(t)$ is added for simplify the presentation. The weight $w(t)$ reflects *accuracy* of the tuple t ; it is assumed that data in tuples $\{t_1, t_2, t_5, t_6\}$ are more accurate than in $\{t_3, t_3, t_7\}$.

We assume that the distance function $dist(v_1, v_2)$ is the string edit distance, which is defined as the minimum number of single-character insertions, deletions and substitutions required to transform v_1 to v_2 . Thus, the cost of modifying the values “555-8195” and “NJ” of the tuple t_3 to “555-8145” and “NY” of tuple t_2 , respectively, is $cost(t_3) = 1 \cdot (1 + 1) = 2$. Conversely, the cost of modifying the values “555-8145” and “NY” of tuple t_2 to “555-8195” and “NJ” of the tuple t_3 , respectively, is $cost(t_2) = 2 \cdot (1 + 1) = 4$.

□

The *cost of a repair* R for a database D is defined as

	<i>PhoneNumber</i>	<i>Name</i>	<i>Street</i>	<i>City</i>	<i>State</i>	<i>Zip</i>	$w(t)$
t_1	949 – 1212	<i>Smith</i>	<i>Bridge</i>	<i>Midville</i>	<i>AZ</i>	05211	2
t_2	555 – 8145	<i>Jones</i>	<i>Valley</i>	<i>Centre</i>	<i>NY</i>	10012	2
t_3	555 – 8195	<i>Jones</i>	<i>Valley</i>	<i>Centre</i>	<i>NJ</i>	10012	1
t_4	212 – 6040	<i>Blake</i>	<i>Mountain</i>	<i>Davis</i>	<i>CA</i>	07912	1

Customer

	<i>SerialNumber</i>	<i>PhoneNumber</i>	<i>Manufacturer</i>	<i>Model</i>	$w(t)$
t_5	<i>AC13006</i>	949 – 1212	<i>AC</i>	<i>XE5000</i>	2
t_6	<i>L55001</i>	555 – 8145	<i>LU</i>	<i>ZE400</i>	2
t_7	<i>L55001</i>	555 – 8195	<i>LU</i>	<i>ZE400</i>	1

*Equipment***Fig. 5.8.** Relations *Customer* and *Equipment*

$$cost(R) = \sum_{t \in R} cost(t)$$

Formally, the problem addressed in [17] is: given a database instance D and a set IC of both functional dependencies and inclusion dependencies, find the repair R for D w.r.t IC such that $cost(R)$ is minimum.

In order to obtain a repair, different actions can be performed depending on the constraint violations. Let t_1 and t_2 be two tuples of the relation P in the database D , and $F = A \rightarrow B$ a functional dependency over P . If t_1 and t_2 violates F , i.e. $D(t_1, A) = D(t_2, A)$ and $D(t_1, B) \neq D(t_2, B)$, then this constraint violation is resolved by setting the B -attribute value of t_1 to be equal to that of t_2 (or vice versa).

Similarly, also violations of inclusion dependencies can be repaired by modifying attribute values. Let t_1 be a tuple in the relation P that does not satisfy the inclusion dependency $P[A] \subseteq Q[B]$. Then we can modify the A -attribute value of t_1 so that it is equal to the B -attribute value for some tuple t_2 in the relation Q . Alternately, the B -attribute value for some tuple t_2 in Q can be modified so that it is equal to the A -attribute value of t_1 .

Moreover, if no similar $Q[B]$ value exists for some unmatched tuple t_1 from P , then inserting a new tuple t' in Q may be preferable to modifying t (this choose depends on which action produces a minimal-cost repair) In this case, the B -attribute values of t' are set to match the attribute values of t , and all other attribute values are set to the special value *null*.

Example 5.9 Consider the Example 5.8. The violations of the functional dependency (ii) can be resolved modifying the value of *PhoneNumber* from “555-8195” of the tuple t_3 to “555-8145” of t_2 . Similarly, the violations of the

functional dependency (iii) can be resolved modifying the value of *State* form “NY” of t_3 to “NJ” of t_2 . Finally, the violation of the key dependency (iv) and the inclusion dependency can be resolved changing the value “555-8145” of t_7 into “555-8195”.

The set of value updates described above give raise to a repair for the database in Figure 5.8 w.r.t. the constraints of Example 5.8. \square

In [17] it has been shown that, given a database D and a set of constraints IC consisting of either only functional dependencies or inclusion dependencies, then for a constant W , the problem of determining if there exists a repair R of D w.r.t. IC whose cost is at most W is *NP*-complete.

5.3.2 A Greedy Algorithm Based on Equivalence Classes

In [17] heuristic approaches has been considered for finding repairs. An heuristic algorithm takes as input a database D and a set of constraints IC , and finds a repair R for D w.r.t. IC . It finds efficiently a solution but with the tradeoff that it is not necessarily minimum (the key difficulty is that repairing one constraint can break another constraint).

The approach works in presence of both functional dependencies and inclusion dependencies. It is built around the notion of *equivalence classes* of attribute value coordinates $\langle t, A \rangle$, where t identifies a tuple in a relation P in which A is an attribute. The semantics of an equivalence class of pairs $\langle t, A \rangle$ is that the attributes of the tuple contained in the class are assigned the same value in the repair R .

Both functional dependencies and full inclusion dependencies can be seen as specifying equivalence between certain sets of attribute coordinates. Specifically, the functional dependencies $V \rightarrow W$ over the relation P specifies that if a pair of tuples t_1 and t_2 in P matches on the attribute set V , then $\langle t_1, A \rangle$ and $\langle t_2, A \rangle$ must be in the same equivalence class for all A in W . Similarly, the inclusion dependency $P[V] \subseteq Q[W]$ requires that each tuple t_1 in P is covered by some tuple t_2 in Q , that is $\langle t_1, V \rangle$ and $\langle t_2, W \rangle$ must be in the same equivalence class for each attribute A in V and the corresponding attribute B in W .

An equivalence class eq is a set of pairs $\langle t, A \rangle$. The repair algorithm maintains a global set of equivalence classes that covers the repair R (that is, the tuples in the original database D , plus insertions). Associated with each class eq is a *target value* $v = targ(eq)$. In a repair R , all attributes in a class eq are assigned the value of $targ(eq)$, that is $R(t, A) = targ(eq)$ for each $\langle t, A \rangle \in eq$.

The cost of the equivalence class for a particular target value v is defined as the contribution of elements in the equivalence class to the cost of the repair R (ignoring the cost of inserts). That is,

$$cost(eq, v) = \sum_{\langle t, A \rangle \in eq} w(t) \cdot dist(D(t, A), v)$$

Consistent with the goal of finding a low-cost repair, $v = \text{targ}(eq)$ is chosen to minimize the cost of eq . The cost of an equivalence class eq , denoted as $\text{cost}(eq)$, is the minimum $\text{cost}(eq, v)$ over the values v taken by the elements of eq in the original database D .

Example 5.10 Consider the database and the constraints of the Example 5.8. The functional dependency $\text{Name}, \text{Street}, \text{Zip} \rightarrow \text{PhoneNumber}$ entails that the *PhoneNumber* of the tuples t_2 and t_3 belongs to the same equivalence class, i.e. there is $eq = \{\langle t_2, \text{PhoneNumber} \rangle, \langle t_3, \text{PhoneNumber} \rangle\}$. It holds that $\text{cost}(eq, \text{"555-8145"}) = 1 \cdot 1$, whereas $\text{cost}(eq, \text{"555-8195"}) = 2 \cdot 1$. Thus the value $\text{targ}(eq)$ is "555-8145" since $\text{cost}(eq, \text{"555-8145"})$ is the minimum cost (among the cost of the possible target values). Finally, $\text{cost}(eq) = 1$. \square

Whenever two equivalence classes are merged, this may result in additional attribute modifications in the repair R increasing its cost. For a subset E of equivalence classes, the *increase in cost* is the difference between the cost of the merged class and the sum of the costs of the individual classes, that is

$$\text{mgcost}(E) = \text{cost}\left(\bigcup_{eq \in E} eq\right) - \sum_{eq \in E} \text{cost}(eq)$$

Example 5.11 Consider the equivalence class $eq_1 = \{\langle t_2, \text{PhoneNumber} \rangle, \langle t_3, \text{PhoneNumber} \rangle\}$ of the Example 5.10. Let eq_2 be the equivalence class $\{\langle t_7, \text{PhoneNumber} \rangle\}$. The increase in cost of merging eq_1 and eq_2 in order to form $eq_3 = \{\langle t_2, \text{PhoneNumber} \rangle, \langle t_3, \text{PhoneNumber} \rangle, \langle t_7, \text{PhoneNumber} \rangle\}$ is given by $\text{mgcost}(\{eq_1, eq_2\}) = \text{cost}(eq_3) - (\text{cost}(eq_1) + \text{cost}(eq_2)) = 2 - (1 + 0) = 1$. Observe that, $\text{cost}(eq_2) = 0$ because eq_2 contains only an item with value v and the cost $\text{cost}(eq, v) = 0$ because of distance between v and itself is zero. \square

The repair R is constructed by resolving violations of functional and inclusion dependencies, which corresponds to merging the appropriate equivalence classes.

In the following, we will denote as $eq(t, A)$ the *current* equivalence class containing $\langle t, A \rangle$. Let $F = V \rightarrow W$ be a functional dependencies over the relation P . Given a repair R , a tuple t_1 of P is *resolved* w.r.t. F if, for all other tuples t_2 of P , either $R(t_1, A) \neq R(t_2, A)$ for some $A \in V$, or for every $B \in W$, $eq(t_1, B) = eq(t_2, B)$. Observe that, a tuple can become *unresolved* w.r.t. F due to a change in the target value of an attribute in V for some other tuple in the repair R (this may happen due to a change in the target value when equivalence classes merge).

Let $I = P[V] \subseteq Q[W]$ be an inclusion dependency. A tuple t_1 of P is *resolved* w.r.t. I if there is some tuple t_2 of Q such that $eq(t_1, A) = eq(t_2, B)$ for every pair of corresponding attributes $A \in V$ and $B \in W$. Observe that,

a tuple resolved w.r.t. I will not become unresolved with respect to other inclusion dependencies (but it can become unresolved w.r.t. some functional dependencies).

Basically, resolving tuples w.r.t. an integrity constraint means merging equivalence classes, and the cost of this operation is that of merging the equivalence classes. When tuples are resolved w.r.t. an inclusion dependency and the cost $insCost$ of inserting a new tuple is less than that of merging equivalence classes, then a new tuple is inserted with cost $insCost$.

Example 5.12 Consider the database and the constraints of the Example 5.8. The functional dependency $Name, Street, Zip \rightarrow PhoneNumber$ entails that the $PhoneNumber$ of the tuples t_2 and t_3 belongs to the same equivalence class. Thus the classes $eq_1 = \{ \langle t_2, PhoneNumber \rangle \}$ and $eq_2 = \{ \langle t_3, PhoneNumber \rangle \}$ must be merged obtaining $eq_3 = \{ \langle t_2, PhoneNumber \rangle, \langle t_3, PhoneNumber \rangle \}$. As seen in Example 5.11, the cost of this operation is $mgcost(\{eq_1, eq_2\}) = 1$.

Similarly, in order to resolve the tuple t_7 w.r.t. the inclusion dependency $Equipment[PhoneNumber] \subseteq Customer[PhoneNumber]$ the classes eq_3 and $eq_4 = \{ \langle t_6, PhoneNumber \rangle, \langle t_7, PhoneNumber \rangle \}$ must be merged. □

The algorithm which heuristically constructs a repair begins by putting each pair $\langle t, A \rangle$ in its own equivalence class, that is a class $\{ \langle t, A \rangle \}$ is created for each pair of tuple and attribute in the database. Then it greedily merges the equivalence classes until all constraints are satisfied.

The approach consists in *resolving* (unresolved) tuples one at time, until no unresolved tuples remains. We denote as $unResolved(ic)$ the set of tuple that *may violates* the integrity constraint ic , that is $unResolved(ic)$ contains *potentially* unresolved tuples w.r.t. ic . The repair algorithm ensures that $unResolved(ic)$ satisfies the following two invariants:

- i) if t is unresolved w.r.t. $I = P[V] \subseteq Q[W]$, then $t \in unResolved(I)$;
- ii) if t is unresolved w.r.t. $F = V \rightarrow W$ defined over the relation P , then $t' \in unResolved(F)$, where t' is some tuple in P such that t' matches t on attributes in V ; the tuple t' serves as proxy for t , and when t' is resolved then also t will be resolved.

At each iteration of the algorithm a tuple to be resolved w.r.t. a constraint ic is selected from $unResolved(ic)$. The resolution of a tuple is achieved by either (i) merging appropriate equivalence classes so that the (locally) minimum increasing cost is obtained, or (ii) inserting new tuple if it is (locally) less onerous than merging.

Tuples are added to $unResolved(ic)$, for some ic , only when new tuple are inserted into the repair or equivalence classes are merged. Observe that, the number of inserted tuples is bounded by the number of equivalence classes, which is bounded by the number of possible distinct values in the database, i.e.

$|D| \cdot \alpha$, where α is the maximum number of attributes in a relation. Moreover, the number of merge events is bounded by the number of equivalence classes.

The algorithm terminates when there are no tuples to be resolved, and the proposed repair R is produced by inserting the new tuples and replacing $\langle t, A \rangle$ values in the original databases D , with $\text{targ}(eq(t, A))$.

At each iteration a tuple and a constraint are chosen for the resolving task. The arbitrary selection of what tuple and constraint to address represents a degree of freedom for designing the equivalence-class-based technique. Two greedy approaches have been present for picking the unresolved tuples to be processed. The former picks an unresolved tuple to repair with minimum resolution cost. The latter give precedence to fixing tuples that are unresolved w.r.t. functional dependencies. An experimental study of the heuristics proposed has been conducted on synthetic and real-world data showing the effectiveness of the approach.

5.4 Querying Inconsistent Databases by Means of Nuclei

In [76, 77, 78] a value-based repairing technique consisting in a database transformation has been proposed. The intuition is that an inconsistent database is transformed in such a way that the subsequent queries on the transformed database retrieve exactly the consistent answer. That is, given a satisfiable set of constraints IC and a relation I , apply a database transformation h_{IC} such that for every query Q , $Q(h_{IC}(I))$ yields exactly the consistent answer to Q on input I and IC . Observe that $h_{IC}(I)$ is not necessarily a repair for I w.r.t. IC , and can be thought as a *condensed representation* of all possible repairs for I w.r.t. IC that is sufficient for consistent query answering.

In [78] it has been shown that for *full dependencies* and *conjunctive queries*, all repairs can be “summarized” into a single tableau, called *nucleus*, such that the consistent answer to any conjunctive query can be obtained by executing the query on the nucleus. A nucleus G is homomorphic to all repairs and it is maximal in the sense that any other tableau that is homomorphic to all repairs, is also homomorphic to G .

Repairs are defined according to change values attribute instead of entire tuples. The definition of (value-based) repairs is given in two steps considering homomorphisms between tableaux (relations containing constants and variables). Informally, given an inconsistent relation I , first, *fixes* are obtained replacing erroneous values by variables. Each fix is homomorphic to the original relation I . Secondly, each fix is homomorphic to a consistent relation obtained substituting constants to the variables (which represent erroneous values). This consistent relation is a *repair* for I .

In the following we first provide an example, then formally present *fixes*, *repairs* and *nuclei*.

Example 5.13 Consider the database in Figure 5.9 storing the price range of different car variants; possible price ranges are: *bottom*, *lower*, *medium*, *upper*. Assume that the four full dependencies in Figure 5.10 are defined.

<i>Model</i>	<i>Version</i>	<i>PriceRange</i>
<i>sedan</i>	<i>luxury</i>	<i>upper</i>
<i>sedan</i>	<i>standard</i>	<i>lower</i>

Fig. 5.9. Cars Database

τ_1 <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th><i>Model</i></th> <th><i>Version</i></th> <th><i>PriceRange</i></th> </tr> </thead> <tbody> <tr> <td><i>x</i></td> <td><i>y</i></td> <td><i>upper</i></td> </tr> <tr> <td><i>x</i></td> <td><i>standard</i></td> <td><i>medium</i></td> </tr> </tbody> </table>	<i>Model</i>	<i>Version</i>	<i>PriceRange</i>	<i>x</i>	<i>y</i>	<i>upper</i>	<i>x</i>	<i>standard</i>	<i>medium</i>	ϵ_1 <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th><i>Model</i></th> <th><i>Version</i></th> <th><i>PriceRange</i></th> </tr> </thead> <tbody> <tr> <td><i>x</i></td> <td><i>y</i></td> <td><i>upper</i></td> </tr> <tr> <td colspan="2" style="text-align: center;"><i>y = luxury</i></td> <td></td> </tr> </tbody> </table>	<i>Model</i>	<i>Version</i>	<i>PriceRange</i>	<i>x</i>	<i>y</i>	<i>upper</i>	<i>y = luxury</i>								
<i>Model</i>	<i>Version</i>	<i>PriceRange</i>																							
<i>x</i>	<i>y</i>	<i>upper</i>																							
<i>x</i>	<i>standard</i>	<i>medium</i>																							
<i>Model</i>	<i>Version</i>	<i>PriceRange</i>																							
<i>x</i>	<i>y</i>	<i>upper</i>																							
<i>y = luxury</i>																									
ϵ_2 <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th><i>Model</i></th> <th><i>Version</i></th> <th><i>PriceRange</i></th> </tr> </thead> <tbody> <tr> <td><i>x</i></td> <td><i>y</i></td> <td><i>z</i></td> </tr> <tr> <td><i>x</i></td> <td><i>y</i></td> <td><i>z'</i></td> </tr> <tr> <td colspan="2" style="text-align: center;"><i>z = z'</i></td> <td></td> </tr> </tbody> </table>	<i>Model</i>	<i>Version</i>	<i>PriceRange</i>	<i>x</i>	<i>y</i>	<i>z</i>	<i>x</i>	<i>y</i>	<i>z'</i>	<i>z = z'</i>			ϵ_3 <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th><i>Model</i></th> <th><i>Version</i></th> <th><i>PriceRange</i></th> </tr> </thead> <tbody> <tr> <td><i>x</i></td> <td><i>luxury</i></td> <td><i>z</i></td> </tr> <tr> <td><i>x</i></td> <td><i>y</i></td> <td><i>bottom</i></td> </tr> <tr> <td colspan="2" style="text-align: center;"><i>0 = 1</i></td> <td></td> </tr> </tbody> </table>	<i>Model</i>	<i>Version</i>	<i>PriceRange</i>	<i>x</i>	<i>luxury</i>	<i>z</i>	<i>x</i>	<i>y</i>	<i>bottom</i>	<i>0 = 1</i>		
<i>Model</i>	<i>Version</i>	<i>PriceRange</i>																							
<i>x</i>	<i>y</i>	<i>z</i>																							
<i>x</i>	<i>y</i>	<i>z'</i>																							
<i>z = z'</i>																									
<i>Model</i>	<i>Version</i>	<i>PriceRange</i>																							
<i>x</i>	<i>luxury</i>	<i>z</i>																							
<i>x</i>	<i>y</i>	<i>bottom</i>																							
<i>0 = 1</i>																									

Fig. 5.10. Full Dependencies $\tau_1, \epsilon_1, \epsilon_2, \epsilon_3$

The first full dependency τ_1 is a *full tuple-generating dependency* saying that every model in the *upper* price range also exists in a *medium* priced *standard* version. The full dependencies ϵ_1, ϵ_2 and ϵ_3 are *full equality-generating dependencies*. ϵ_1 says that the *upper* price range only contains *luxury* cars. ϵ_2 is a key dependency: the price range of a car is determined by its model and version. ϵ_3 is a *contradiction-generating dependency* stating that models which exist in a *luxury* version are never available at *bottom* prices.

The relation shown in Figure 5.9 falsifies τ_1 , because it does not contain the tuple $\langle \textit{sedan}, \textit{standard}, \textit{medium} \rangle$. Adding that tuple results in a violation of the key dependency (ϵ_2).

Five *fixes* $F_1 - F_5$ of the relation of Figure 5.9 are shown in Figure 5.11. The fix F_1 assumes that the value “sedan” in the first tuple is mistaken. F_1 is homomorphic to the original relation (one can substitute the variable x with the constant “sedan”). The fix F_2 assumes that the value “upper” in the first tuple is mistaken. Similarly, the fixes F_3, F_4 and F_5 assume that the values “sedan”, “standard” and “lower” in the second tuple are mistaken, respectively.

Each fix is homomorphic to a consistent relation. In fact, each fix F_i ($1 \leq i \leq 5$) in Figure 5.11 is homomorphic to T_i , and to every relation obtained from T_i by substituting a constant for the variable (x, y or z) present in it. If

we choose a constant distinct from “sedan” for the variable x in T_1 , then the relation obtained will be a *repair*. Similarly, if we choose a constant distinct from “upper” for the variable z in T_2 , we obtain a *repair*. Analogously for T_3 , T_4 and T_5 .

F_1	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>x</td><td>luxury</td><td>upper</td></tr><tr><td>sedan</td><td>standard</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	x	luxury	upper	sedan	standard	lower
Model	Version	PriceRange								
x	luxury	upper								
sedan	standard	lower								
F_2	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>z</td></tr><tr><td>sedan</td><td>standard</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	z	sedan	standard	lower
Model	Version	PriceRange								
sedan	luxury	z								
sedan	standard	lower								
F_3	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>upper</td></tr><tr><td>x</td><td>standard</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	upper	x	standard	lower
Model	Version	PriceRange								
sedan	luxury	upper								
x	standard	lower								
F_4	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>upper</td></tr><tr><td>sedan</td><td>y</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	upper	sedan	y	lower
Model	Version	PriceRange								
sedan	luxury	upper								
sedan	y	lower								
F_5	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>upper</td></tr><tr><td>sedan</td><td>standard</td><td>z</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	upper	sedan	standard	z
Model	Version	PriceRange								
sedan	luxury	upper								
sedan	standard	z								

T_1	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>x</td><td>luxury</td><td>upper</td></tr><tr><td>x</td><td>standard</td><td>medium</td></tr><tr><td>sedan</td><td>standard</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	x	luxury	upper	x	standard	medium	sedan	standard	lower
Model	Version	PriceRange											
x	luxury	upper											
x	standard	medium											
sedan	standard	lower											
T_2	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>z</td></tr><tr><td>sedan</td><td>standard</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	z	sedan	standard	lower			
Model	Version	PriceRange											
sedan	luxury	z											
sedan	standard	lower											
T_3	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>upper</td></tr><tr><td>sedan</td><td>standard</td><td>medium</td></tr><tr><td>x</td><td>standard</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	upper	sedan	standard	medium	x	standard	lower
Model	Version	PriceRange											
sedan	luxury	upper											
sedan	standard	medium											
x	standard	lower											
T_4	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>upper</td></tr><tr><td>sedan</td><td>standard</td><td>medium</td></tr><tr><td>sedan</td><td>y</td><td>lower</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	upper	sedan	standard	medium	sedan	y	lower
Model	Version	PriceRange											
sedan	luxury	upper											
sedan	standard	medium											
sedan	y	lower											
T_5	<table border="1"><thead><tr><th>Model</th><th>Version</th><th>PriceRange</th></tr></thead><tbody><tr><td>sedan</td><td>luxury</td><td>upper</td></tr><tr><td>sedan</td><td>standard</td><td>medium</td></tr></tbody></table>	Model	Version	PriceRange	sedan	luxury	upper	sedan	standard	medium			
Model	Version	PriceRange											
sedan	luxury	upper											
sedan	standard	medium											

Fig. 5.11. Fixes $F_1 - F_5$, and Tableaux $T_1 - T_5$

A *nucleus*, which is homomorphic to all repairs, for the database in Figure 5.9 and the full functional dependencies in Figure 5.10, is shown in Figure 5.12. Answers obtained querying the nucleus are the consistent answers which are obtained querying all the repairs (which are in general large in number or infinite).

□

5.4.1 Tableaux Formalism

We introduce the tableau formalism which will be exploited for formally defining the repair framework. To simplify the notation, we will assume a unirelational database containing a single relation of arity n .

<i>Model</i>	<i>Version</i>	<i>PriceRange</i>
x_1	<i>luxury</i>	x_2
x_1	<i>standard</i>	x_3
<i>sedan</i>	<i>standard</i>	x_4

Fig. 5.12. Nucleus for the Cars Database

A *tableau* is a relation that can contain variables, that is for each tuple $u = \langle w_1, \dots, w_n \rangle$ in a tableau, w_i is a constant or a variable.

A substitution is a mapping θ from variables to symbols, extended to be the identity on constants. We write id for the identity function on symbol; and write $id_{p=q}$, where p and q are not two distinct constants, for a substitution that identifies p and q and that is the identity otherwise. That is if p is a variable and q a constant, then $id_{p=q} = \{p/q\}$. If p and q are variables, then $id_{p=q}$ can be either $\{p/q\}$ or $\{q/p\}$.

Substitutions naturally extend to tuples and tableaux: firstly, $\theta(\langle w_1, \dots, w_n \rangle) = \langle \theta(w_1), \dots, \theta(w_n) \rangle$, and secondly if T is a tableau, then $\theta(T) = \{\theta(u) \mid u \in T\}$.

We use the tableau formalism to express conjunctive queries. A *tableau query* is a pair (B, h) where B is a tableau (called *body*) and h is a tuple (called *summary* or *head*) such that every variable in h also occurs in B ; B and h need not have the same arity. Let $Q = (B, h)$ be a tableau query, and T a tableau of the same arity as B . A tuple t is an *answer* to Q on input T iff there exists a substitution θ for the variables in B such that $\theta(B) \subseteq T$ and $\theta(h) = t$. The set of all answers to Q on input T is denoted $Q(T)$.

Let F, G be two tableau of fixed arity n . A *homomorphism* from F to G is a substitution θ for the variables in F such that $\theta(F) \subseteq G$. If such homomorphism from F to G exists, then F is said to be *homomorphic* to G , denoted $G \succeq F$. Two tableaux are said to be *equivalent*, denoted as $F \sim G$ iff $G \succeq F$ and $F \succeq G$. We write $G \succ F$ iff $G \succeq F$ and $F \not\succeq G$. The relation \sim is an equivalence relation.

The relation \sim on tableaux naturally extends to sets of tableaux: two sets \mathbf{F} and \mathbf{G} of tableaux are said to be equivalent, denoted $\mathbf{F} \sim \mathbf{G}$, iff for every tableau in either set, there exists an equivalent tableau in the other set.

A *full dependency* is either a full *tuple-generating dependency* (*ftgd*) or a *full equality-generating dependency* (*fegd*). Satisfaction of full dependencies by relations is borrowed from first-order logic semantics. We also define what it means for a tableau to satisfy a set of full dependencies.

A *ftgd* takes the form of a tableau query (B, h) where B and h have the same arity. The *ftgd* $\tau = (B, h)$ is satisfied by a tableau T , denoted $T \models \tau$, iff $T \cup \tau(T) \sim T$. A *fegd* is of the form $(B, p = q)$ where B is a tableau and p and q are symbols such that every variable in $\{p, q\}$ also occurs in B . The *fegd* $\epsilon = (B, p = q)$ is satisfied by a tableau T , denoted $T \models \epsilon$, iff for every

substitution θ , if $\theta(B) \subseteq T$ then $\theta(p)$, $\theta(q)$ are not two distinct constants and $T \sim id_{\theta(p)=\theta(q)}(T)$.

A *contradiction-generating dependency* (*cgd*) is an fegd of the form $(B, a = b)$, where a and b are distinct constants.

A tableau F is called consistent w.r.t. a set of full dependencies IC , denote as $F \models IC$, iff for each $ic \in IC$, $F \models ic$.

It was shown that given two tableaux F and G and a set of full dependencies IC , if $F \sim G$ and $F \models IC$, then $G \models IC$.

5.4.2 Fixes and Repairs

Consistency of relations is defined relative to a set IC of integrity constraints. A tableau T will be called *subconsistent* w.r.t. IC iff T is homomorphic to a consistent relation, i.e. $J \succeq T$ for some relation J such that $J \models IC$.

Intuitively, a *fix* of a relation instance I is a subconsistent tableau obtained from I by replacing erroneous values by distinct variables.

Let F, G be two tableau of fixed arity n . A *one-one homomorphism* from F to G is an homomorphism from F to G that do not identify distinct tuples. That is, it is a substitution θ for the variables in F such that $\theta(F) \subseteq G$ and for all $u, u' \in F$, $u \neq u'$ implies $\theta(u) \neq \theta(u')$. If such one-one homomorphism from F to G exists, then F is said to be *one-one homomorphic* to G , denoted $G \sqsupseteq F$. We write $F \simeq G$ iff $G \sqsupseteq F$ and $F \sqsupseteq G$. Further, we write $G \sqsupset F$ iff $G \sqsupseteq F$ and $G \not\sqsupseteq F$.

A tuple or tableau without multiple occurrences of the same variable is called *linear*. Let IC be a satisfiable set of constraints and I a relation instance. A *fix* of I w.r.t IC is a *linear* tableau F satisfying:

- i) $I \sqsupseteq F$ and F is *subconsistent*;
- ii) *Maximality*: for every linear tableau G , if $I \sqsupseteq G \sqsupset F$, then G is not subconsistent.

For each relation I and for each set IC which is satisfiable by a nonempty relation, if F is a fix of I w.r.t IC , then $|F| = |I|$ holds.

The set of fixes of I w.r.t IC will be denoted as $\mathbf{F}(I, IC)$.

Example 5.14 Each tableau F_i ($1 \leq i \leq 5$) shown in Figure 5.11 is subconsistent w.r.t. $IC = \{\tau_1, \epsilon_1, \epsilon_2, \epsilon_3\}$. For instance, F_1 is homomorphic to the consistent relations obtained from the tableau T_1 by substituting a constant distinct from “sedan” for the variable x in T_1 . Similar considerations hold for the tableaux F_2, F_3, F_4 and F_5 .

Moreover, it is easy to verify these tableaux are fixes of the relation shown in Figure 5.9 w.r.t. IC . □

In [76] the author allow multiple occurrence of the same variable in a fix (obtaining not linear tableau), but, as shown in [77], considering fixes where no

variable occurs more than once accomplishes the quite natural “independence-of-errors” assumption. Moreover, this restriction gives us tractability of the *fix checking problem* for the class of constraints consisting of full dependencies. That is, given a set of full dependencies IC , a relation I and a linear tableau F of the same arity, deciding whether $F \in \mathbf{F}(I, IC)$ is in *PTIME*. On the other hand, the fix checking problem becomes *NP*-hard if not-linear tableau are considered.

Given a relation instance I and a set of integrity constraints IC , a *repair* for I w.r.t. IC is a consistent relation R ($R \models IC$) such that there is a fix F of I and IC satisfying

- i) $R \succeq F$, and
- ii) for every relation J' , if $R \succ J' \succeq F$, then $J' \not\models IC$.

Observe that, since for each pair of relations G and G' , $G \succ G'$ iff $G' \subset G$, the second condition requires that R is minimal (under \subseteq) among the relations which F is homomorphic to.

Intuitively, in repairing a relation I , we first go down the homomorphism lattice to find fixes, then go up to find repairs.

Example 5.15 Every relation obtained from the tableau T_1 by substituting a constant distinct from “sedan” for the variable x is a repair of the relation shown in Figure 5.9 w.r.t. $IC = \{\tau_1, \epsilon_1, \epsilon_2, \epsilon_3\}$. □

The set repairs for a relation I w.r.t. IC is denote as $\mathcal{R}(I, IC)$. Given a relation I , a set of integrity constraints IC and a tableau query Q , the *consistent query answer* to Q on input I and IC , denoted $CQA(Q, I, IC)$, is the intersection of the answers to Q on all (possibly infinitely many) repair generated by I and IC .

Let I be a relation, Q be a tableau query and IC a satisfiable set of full dependencies. It has been shown that the set of consistent query answers $CQA(Q, I, IC)$ can be obtained by *chasing* each fix in $\mathbf{F}(I, IC)$.

Chasing Fixes

The *chase*, originally introduced for deciding logical implication is used for repairing databases. In particular, some results of [14] are generalized to tableaux that can contain constants and need not be typed, replacing the equality of tableaux by equivalence relation \sim .

An artificial top element, denoted \square , is introduced to the quasi-order $\langle \mathbf{T}, \succeq \rangle$, where \mathbf{T} is the set of all tableaux of fixed arity n . Let $F \neq \square$ and G be tableaux and let IC be a set of full dependencies. We write $F \vdash_{IC} G$ if G can be obtained from F by a single application of one of the following *chase rules*:

- If $Q = (B, h)$ is a *ftgd* of IC , then $F \vdash_{IC} F \cup Q(F)$.

- Let $(B, p = q)$ be a *fgd* of IC , and θ a substitution such that $\theta(B) \subseteq F$. If $\theta(p)$ and $\theta(q)$ are two distinct constants, then $F \vdash_{IC} \square$; otherwise, $F \vdash_{IC} id_{\theta(p)=\theta(q)}(F)$.

A *chase* of a tableau F by IC is a *maximal* (w.r.t. length) sequence $F = F_0, F_1, \dots, F_n$ of tableaux such that for every $i \in \{1, \dots, n\}$, $F_{i-1} \vdash_{IC} F_i$ and $F_i \neq F_{i-1}$.

Requiring that chases be maximal tacitly assumes that chases are finite.

Given a tableau $F \neq \square$ and a set of full dependencies, then the following statements hold:

- If G is a tableau in a chase of F by IC , then $G \succeq F$.
- Each chase of F by IC is finite.
- If $G \neq \square$ is the last element of a chase of F by IC , then $G \models IC$.
- If $G \neq \square$ is the last element of a chase of F by IC , and θ is a substitution mapping distinct variables to new distinct constants not occurring elsewhere, then the relation $\theta(G) \models IC$.

Let $F \neq \square$ be a tableau and IC a set of full dependencies. It was shown that if two chase of F end with G_1 and G_2 respectively, then $G_1 \sim G_2$. We write $\mathbf{chase}(F, IC)$ for the class $[G]_{\sim}$ if the last element of a chase of F by IC is G . The singleton $[\square]_{\sim}$ is also written \square .

Given a set of full dependencies IC , it is decidable whether a given tableau $F \neq \square$ is subconsistent: F is subconsistent if and only if $\mathbf{chase}(F, IC) \neq \square$. Thus, a set of full dependencies IC is satisfiable iff $\mathbf{chase}(\emptyset, IC) \neq \square$. Moreover, let F, G be tableaux, both distinct from \square , and IC a set of full dependencies. If $F \sim G$, then $\mathbf{chase}(F, IC) = \mathbf{chase}(G, IC)$.

An important result is the following. Let I be a relation, IC a satisfiable set of full dependencies, τ a tableau query and $\mathbf{F}(I, IC)$ a finite set of fixes of I and IC . Assume that \mathbf{G} is a set of tableaux such that

$$\mathbf{G} \sim \bigcup_{F \in \mathbf{F}(I, IC)} \mathbf{chase}(F, IC)$$

Then

$$CQA(Q, I, IC) = \bigcap_{G \in \mathbf{G}} \mathbf{grd}(Q(G))$$

where for a tableau T , $\mathbf{grd}(T) = \{t \in T \mid t \text{ is ground}\}$.

Therefore, an effective algorithm to compute $CQA(Q, I, IC)$ is i) chase each tableau of $\mathbf{F}(I, IC)$, and ii) query the last tableau of each chase by Q and return the ground tuples common to all query answers.

Example 5.16 Let $\mathbf{F}(I, IC) = \{F_1, F_2, F_3, F_4, F_5\}$, the set of fixes of relation I shown in Figure 5.9 w.r.t. $IC = \{\tau_1, \epsilon_1, \epsilon_2, \epsilon_3\}$. A chase of F_i by IC ends with T_i ($1 \leq i \leq 5$), which are shown in Figure 5.11. For any tableau query Q , $CQA(Q, I, IC) = \mathbf{grd}(Q(T_1)) \cap \dots \cap \mathbf{grd}(Q(T_5))$.

□

5.4.3 Nuclei and Consistent Query Answers

A nucleus of a (possible infinite) set of relations \mathbf{I} is defined relative to a class of queries; intuitively, it is a single tableau that can replace \mathbf{I} for the purpose of query answering. Let \mathcal{Q} be a subclass of Conjunctive Queries \mathcal{CQ} . The tableau F is called \mathcal{Q} -nucleus of \mathbf{I} if and only if for every query $Q \in \mathcal{Q}$,

$$\mathbf{grd}(Q(F)) = \bigcap_{I \in \mathbf{I}} Q(I)$$

where the function $\mathbf{grd}(\cdot)$ serves to eliminate from $Q(F)$ tuples that contain variables.

A nucleus is unique up to \sim , i.e if G_1 and G_2 are \mathcal{CQ} -nuclei of a set \mathbf{I} , then $G_1 \sim G_2$.

Given a finite set of tableau \mathbf{T} , the *greatest lower bound* of \mathbf{T} w.r.t. \succeq is a tableau G satisfying:

- i) for each $F \in \mathbf{T}$, $F \succeq G$, and
- ii) for every tableau G' , if for each $F \in \mathbf{T}$, $F \succeq G'$, then $G \succeq G'$.

All greatest lower bounds of a set of tableaux \mathbf{T} are mutually equivalent.

A main result is the following. Let I be a relation, IC a satisfiable set of full dependencies and $\mathbf{F}(I, IC)$ a finite set of fixes of I and IC . Assume that \mathbf{G} is a set of tableaux such that $\mathbf{G} \sim \bigcup_{F \in \mathbf{F}(I, IC)} \mathbf{chase}(F, IC)$. Then every greatest lower bound of \mathbf{G} is i) consistent, and ii) a \mathcal{CQ} -nucleus of $\mathcal{R}(I, IC)$ (the set of repairs for I w.r.t. IC).

Therefore, an effective procedure for computing a \mathcal{CQ} -nucleus of $\mathcal{R}(I, IC)$ is chase each fix F in $\mathbf{F}(I, IC)$ and then compute the greatest lower bound of the last tableaux of these chase.

Example 5.17 Let $\mathbf{F}(I, IC) = \{F_1, F_2, F_3, F_4, F_5\}$, the set of fixes of relation I shown in Figure 5.9 w.r.t. $IC = \{Q_1, \epsilon_1, \epsilon_2, \epsilon_3\}$. A chase of F_i by IC ends with T_i ($1 \leq i \leq 5$), which are shown in Figure 5.11. The greatest lower bound G of the set of tableaux $\{T_1, T_2, T_3, T_4, T_5\}$ is shown in Figure 5.12. For any tableau query Q , $Q(G)$ gives us exactly the consistent answers to Q on the relation I w.r.t. IC . □

Let IC be a set of full dependencies. On input of a relation I , we can effectively compute a \mathcal{CQ} -nucleus of $\mathcal{R}(I, IC)$. This nucleus allows us to compute consistent answers to any tableau query. However, the nucleus may not be practical because its construction takes exponential time, or even worse, its size is exponential. Construction of nuclei (and hence consistent query answering) is tractable for restricted classes of conjunctive and full dependencies.

For a database D , if IC is a set of full dependencies and Q is a tableau query, then consistent query answer problem $\mathcal{CQA}(Q, D, IC)$ is *coNP*-complete. Thus unless $P=NP$, the construction of a \mathcal{CQ} -nucleus takes exponential time.

Moreover, in [78] it has been shown that, for full dependencies and conjunctive queries, the size of nuclei may be not polynomially bounded: there exists a set of full dependencies such that for every $n > 7$, there exists a relation with $n - 1$ tuples allowing no \mathcal{CQ} -nucleus of size less than 2^n .

We now introduce restricted query classes that allow nuclei of polynomial size. The query classes are obtained from the class of Conjunctive Queries by limiting the number of occurrences of quantified variables.

A tableau query $Q = (B, h)$ is said to be *linear* if every variable that occurs more than once in B also occurs in h . The class of linear tableau queries is denoted $\text{lin}\mathcal{CQ}$.

Example 5.18 A linear tableau query $Q = (B, h)$ is shown in Figure 5.13. It asks for the models that exist in both *upper* and *bottom* priced versions. Note that (B, h) linear does not imply that the tableau B is linear.

Q	<i>Model</i>	<i>Version</i>	<i>PriceRange</i>
	x	y	<i>upper</i>
	x	z	<i>bottom</i>
	x		

Fig. 5.13. A linear tableau query

□

Quantifier-free tableau queries further restrict linear tableau queries. A tableau query $Q = (B, h)$ is said to be *quantifier-free* if every variable that occurs in B also occurs in h . The class of quantifier-free tableau queries is denoted $\text{qf}\mathcal{CQ}$. Obviously, $\text{qf}\mathcal{CQ} \subset \text{lin}\mathcal{CQ} \subset \mathcal{CQ}$.

In [78] it has been shown that, for any relation I and satisfiable set IC of full dependencies,

1. there exists a *lin* \mathcal{CQ} -nucleus of $\mathcal{R}(I, IC)$ which is a *linear* tableau whose size is polynomially bounded in $|I|$.
2. there exists a *qf* \mathcal{CQ} -nucleus of $\mathcal{R}(I, IC)$ which is a *ground* tableau and whose size is polynomially bounded in $|I|$.

Given a relation I with *one key dependency* KD , constructing a *lin* \mathcal{CQ} -nucleus of $\mathcal{R}(I, KD)$ takes $O(m \log m)$ time, where $m = |I|$. Thus, consistent query answer problem is tractable for relations with at most one key constraint. If we omit the linearity restriction on the query, consistent query answer problem becomes *coNP*-complete.

Given a relation I and a set IC of *contradiction-generating dependencies*, a *qf* \mathcal{CQ} -nucleus of $\mathcal{R}(I, IC)$ can be computed in polynomial time in $|I|$. Consistent query answering becomes *coNP*-complete for linear tableau queries.

For different classes of constraints and queries, the complexity results for the consistent query answer problem are summarized in Table 5.1. The *PTIME* cases also have nucleus constructible in polynomial time.

		Class of Queries		
		<i>qfCQ</i>	<i>linCQ</i>	<i>CQ</i>
<i>IC</i>	<i>1 key dependency</i>	<i>PTIME</i>	<i>PTIME</i>	coNP-complete
	<i>cgds</i>	<i>PTIME</i>	coNP-complete	coNP-complete
	<i>full dependencies</i>	coNP-complete	coNP-complete	coNP-complete

Table 5.1. Complexity results for the consistent query answer problem

5.5 Discussion

In this chapter we have discussed several attribute-based techniques for computing repairs for a database. The main difference with the techniques examined in Chapter 3 and Chapter 4 is that, here, changes in attribute values are considered as basic repair actions. The first repairing approach that does not treat tuples as atomic unit of repairing is that of [42] (cfr. Section 5.1), where a repair is defined as a mapping from a possibly inconsistent relation to a new consistent relation. This notion of repair is very similar to that formalized in [16] (cfr. Section 5.2), where consistency of a database is achieved by changing only some *fixable* attribute values and keeping the values for the attributes in the key of the relations. In [42] the DLV system [37] has been exploited for computing *preferred repairs* w.r.t. first-order constraints which express *edit rules* of a *questionnaire* collecting census data. In [16] problem of repairing databases by changing integer numerical values has been studied, and *LS-repairs* have been defined. An LS-repair for a database D is a consistent database instance minimizing the square of Euclidean distance between the integer values in D and modified integer values in the repair for D . The problem of repairing numerical database with respect to denial constraints has been investigated. After some intractability results, it has been shown that for *local* denial constraints, LS-repairs can be computed by transforming this problem into an instance of the *Minimum Weighted Set Cover Optimization Problem*. By concentrating on local denials, an approximation within a constant factor has been provided.

In the two approaches above repairs are obtained by only changing an appropriate subset of attribute values. Thus, a database is repaired by only updating attribute values; neither new tuple are inserted into the original database instance, nor tuple are deleted from the original instance. On the other hand, according to the definition of repairs in [17] it is possible that from two distinct (conflicting) tuples we obtain a single (consistent) tuple

by means of value modification (cfr. Section 5.3). This notion of repair has been motivated by showing a parallelism between searching for a repair for an inconsistent database and the *record linkage* task, where pairs of records that refer to the same entity in different data sets have to be associated. Moreover, in order to restore the consistency also insertions are allowed in presence of inclusion dependencies.

In [76] repairs are obtained by defining an homomorphism lattice. In repairing a database one, first, goes down the homomorphism lattice to find *fixes*, then goes up to find *repairs* (cfr. Section 5.4). Fixes correspond to *tableaux* obtained from inconsistent relations by replacing erroneous values by distinct variables. Whereas, repairs can be obtained by *chasing* fixes. Although the basic primitive used to find a fix is the changing of attribute values, the result of chase may yield repairs such that they consist of inserting and deleting tuples. In [76] it has been shown that for full dependencies and conjunctive queries, repairs for a database can be summarized into a single tableau called *nucleus* for the database. The consistent answers to a conjunctive query can be obtained by executing the query on the nucleus for the given database. We now compare the use of nuclei with the query rewriting approaches [4, 45, 46] (cfr. Section 2.5, Section 3.3 and Section 3.4). We point out that the use of the nucleus for a database eliminates the need of rewriting a given query, since a nucleus can be computed once and then used for answering to any queries. But, a nucleus has to be recomputed when the database is modified. Although the database is often modified, the use of the nucleus for a database is meaning for the class of constraints and queries such that nucleus construction need no more time than the time required to answer a single query obtained from rewriting.

Repairing and Querying Numerical Databases under Aggregate Constraints

In this chapter we investigate the problem of repairing and extracting reliable information from data violating a given set of *aggregate constraints*. These constraints consist of linear inequalities on aggregate-sum queries issued on *measure values* stored in the database. This syntactic form enables meaningful constraints which often occur in practice to be expressed.

The notion of repair as consistent set of updates at attribute-value level is exploited, and the characterization of several data-complexity issues related to repairing data and computing consistent query answers is provided. We adopt two different criteria for determining whether a set of update operations repairing data can be considered “reasonable” or not: *set-minimal* semantics and *card-minimal* semantics. Both these semantics aim at preserving the information represented in the source data as much as possible. They correspond to different repairing strategies which turn out to be well-suited for different application scenarios.

We provide the complexity characterization of three fundamental problems: (i) the repair-existence problem, (ii) the repair checking problem (under both the *set-minimal* and *card-minimal* semantics), and (iii) the consistent query answer problem.

6.1 Introduction

Research has deeply investigated several issues related to the use of integrity constraints on relational databases. In this context, a great deal of attention has been devoted to the problem of extracting reliable information from databases containing pieces of information inconsistent w.r.t. some integrity constraints. As seen in Chapter 3, Chapter 4 and Chapter 5, all previous works in this area deal with “classical” forms of constraint (such as keys, foreign keys, functional dependencies), and propose different strategies for updating inconsistent data reasonably, in order to make it consistent by means of minimal changes. Indeed these kinds of constraint often do not suffice to manage

data consistency, as they cannot be used to define algebraic relations between stored values. In fact, this issue frequently occurs in several scenarios, such as scientific databases, statistical databases, and data warehouses, where numerical values of tuples are derivable by aggregating values stored in other tuples.

In this chapter we focus our attention on databases where stored data violates a set of *aggregate constraints*, i.e. integrity constraints defined on aggregate values extracted from the database. These constraints are defined on numerical attributes (such as sales prices, costs, etc.) which represent measure values and are not intrinsically involved in other forms of constraints.

Example 6.1 Table 6.1 represents a two-years *cash budget* for a firm, that is a summary of cash flows (receipts, disbursements, and cash balances) over the specified periods. Values ‘*det*’, ‘*aggr*’ and ‘*drv*’ in column *Type* stand for *detail*, *aggregate* and *derived*, respectively. In particular, an item of the table is *aggregate* if it is obtained by aggregating items of type *detail* of the same section, whereas a *derived* item is an item whose value can be computed using the values of other items of any type and belonging to any section.

<i>Year</i>	<i>Section</i>	<i>Subsection</i>	<i>Type</i>	<i>Value</i>
2003	Receipts	beginning cash	drv	20
2003	Receipts	cash sales	det	100
2003	Receipts	receivables	det	120
2003	Receipts	total cash receipts	aggr	250
2003	Disbursements	payment of accounts	det	120
2003	Disbursements	capital expenditure	det	0
2003	Disbursements	long-term financing	det	40
2003	Disbursements	total disbursements	aggr	160
2003	Balance	net cash inflow	drv	60
2003	Balance	ending cash balance	drv	80
2004	Receipts	beginning cash	drv	80
2004	Receipts	cash sales	det	100
2004	Receipts	receivables	det	100
2004	Receipts	total cash receipts	aggr	200
2004	Disbursements	payment of accounts	det	130
2004	Disbursements	capital expenditure	det	40
2004	Disbursements	long-term financing	det	20
2004	Disbursements	total disbursements	aggr	190
2004	Balance	net cash inflow	drv	10
2004	Balance	ending cash balance	drv	90

Table 6.1. A cash budget

A cash budget must satisfy the following integrity constraints:

1. for each section and year, the sum of the values of all *detail* items must be equal to the value of the *aggregate* item of the same section and year;
2. for each year, the net cash inflow must be equal to the difference between total cash receipts and total disbursements;
3. for each year, the ending cash balance must be equal to the sum of the beginning cash and the net cash balance.

Table 6.1 was acquired by means of an OCR tool from two paper documents, reporting the cash budget for 2003 and 2004. The original paper document was consistent, but some symbol recognition errors occurred during the digitizing phase, as constraints 1) and 2) are not satisfied on the acquired data for year 2003, that is:

- i) in section *Receipts*, the aggregate value of *total cash receipts* is not equal to the sum of detail values of the same section.
- ii) the value of *net cash inflow* is not to equal the difference between *total cash receipts* and *total disbursements*.

In order to exploit the digital version of the cash budget, a fundamental issue is to define a reasonable strategy for locating OCR errors, and then “repairing” the acquired data to extract reliable information.

□

Most of well-known techniques for repairing data violating either key constraints or functional dependencies accomplish this task by performing deletions and insertions of tuples (cfr. Chapter 3 and Chapter 4). Indeed this approach is not suitable for contexts analogous to that of Example 6.1, that is of data acquired by OCR tools from paper documents. For instance, repairing Table 6.1 by either adding or removing rows means hypothesizing that the OCR tool either jumped a row or “invented” it when acquiring the source paper document, which is rather unrealistic. The same issue arises in other scenarios dealing with numerical data representing pieces of information acquired automatically, such as sensor networks. In a sensor network with error-free communication channels, no reading generated by sensors can be lost, thus repairing the database by adding new readings (as well as removing collected ones) is of no sense. In this kind of scenario, the most natural approach to data repairing is updating directly the numerical data: this means working at attribute-level, rather than at tuple-level. For instance, in the case of Example 6.1, we can reasonably assume that inconsistencies of digitized data are due to symbol recognition errors, and thus trying to re-construct actual data values is well founded. Likewise, in the case of sensor readings violating aggregate constraints, we can hypothesize that inconsistency is due to some trouble occurred at a sensor while generating some reading, thus repairing data by modifying readings instead of deleting (or inserting) them is justified.

6.2 Notations

Given a relation scheme $P(A_1, \dots, A_n)$ we denote as $DOM(A_i)$ the domain of the attribute A_i with $1 \leq i \leq n$. Each $DOM(A_i)$ can be either \mathbb{Z} (infinite domain of integers), \mathbb{Q} (rationals), or \mathbb{S} (strings). The set of attribute names $\{A_1, \dots, A_n\}$ of relation scheme P will be denoted as \mathcal{A}_P .

Domains \mathbb{Q} and \mathbb{Z} will be said to be *numerical domains*, and attributes defined over \mathbb{Q} or \mathbb{Z} will be said to be *numerical attributes*.

Given a relation scheme $P(A_1, \dots, A_n)$, we will denote the set of numerical attributes representing measure data as \mathcal{M}_P (namely, *Measure attributes*). That is, \mathcal{M}_P specifies the set of attributes representing measure values, such as weights, lengths, prices, etc. For instance, in Example 6.1, \mathcal{M}_P consists of the only attribute *Value*. We denote as \mathcal{K}_P the subset of \mathcal{A}_P consisting of the names of the attributes which are a key for P . For instance, in Example 6.1, we have $\mathcal{K}_P = \{Year, Subsection\}$. Throughout this chapter, we assume that $\mathcal{K}_P \cap \mathcal{A}_P = \emptyset$, i.e., measure attributes of a relation scheme P are not used to identify tuples of P . This assumption results in no loss of generality, since in real-world scenarios (such as that considered in Example 6.1) attributes identifying tuples do not contain measure values.

Given a database scheme \mathcal{D} , we will denote as $\mathcal{M}_{\mathcal{D}}$ the union of the sets of measure attributes associated with all the relation schemes in \mathcal{D} .

Given a boolean formula α consisting of comparison atoms of the form $X \diamond Y$, where X, Y are either attributes of P or constants and \diamond is a comparison operator, we say that a tuple t of $P(A_1, \dots, A_n)$ satisfies α (and denote it as $t \models \alpha$) if replacing each occurrence of attribute A_i in α (for each $i \in [1..n]$) with value $t[A_i]$ makes α true.

Given two sets M, M' , $M \Delta M'$ denotes their symmetric difference $(M \cup M') \setminus (M \cap M')$.

6.3 Aggregate Constraints

Given a relation scheme $P(A_1, \dots, A_n)$, an *attribute expression* on P is defined recursively as follows:

- a numerical constant is an attribute expression;
- each numerical attribute A_i (with $i \in [1..n]$) is an attribute expression;
- $e_1 \psi e_2$ is an attribute expression on P , if e_1, e_2 are attribute expressions on P and ψ is an arithmetic operator in $\{+, -\}$;
- $c \times e$ is an attribute expressions on P , if e is an attribute expression on P and c a numerical constant in \mathbb{Q} .
- (e) and $-(e)$ are attribute expressions on P , if e is an attribute expression on P .

Let P be a relation scheme and e an attribute expression on P . An *aggregation function* on P is a function $\chi : (\Delta_1 \times \dots \times \Delta_k) \rightarrow \mathbb{Q}$, where each Δ_i is either \mathbb{Z} , or \mathbb{Q} , or \mathbb{S} , and it is defined as follows:

$$\chi(x_1, \dots, x_k) = \text{SELECT sum}(e) \\ \text{FROM } P \\ \text{WHERE } \alpha(x_1, \dots, x_k)$$

where $\alpha(x_1, \dots, x_k)$ is a boolean formula on x_1, \dots, x_k , constants and attributes of P .

Example 6.2 The following aggregation functions are defined on the relation scheme $CashBudget(Year, Section, Subsection, Type, Value)$ of Example 6.1:

$$\chi_1(x, y, z) = \text{SELECT sum}(Value) \\ \text{FROM CashBudget} \\ \text{WHERE Section} = x \\ \text{AND Year} = y \text{ AND Type} = z$$

$$\chi_2(x, y) = \text{SELECT sum}(Value) \\ \text{FROM CashBudget} \\ \text{WHERE Year} = x \\ \text{AND Subsection} = y$$

Function χ_1 returns the sum of *Value* of all the tuples having *Section* x , *Year* y and *Type* z . For instance, χ_1 (‘Receipts’, ‘2003’, ‘det’) returns $100 + 120 = 220$, whereas χ_1 (‘Disbursements’, ‘2003’, ‘aggr’) returns 160. Function χ_2 returns the sum of *Value* of all the tuples where *Year* = x and *Subsection* = y . In Example 6.1, as the pair *Year, Subsection* is a key for the tuples of *CashBudget*, the sum returned by χ_2 is an attribute value of a single tuple. For instance, χ_2 (‘2003’, ‘cash sales’) returns 100, whereas χ_2 (‘2004’, ‘net cash inflow’) returns 10.

□

Definition 6.1 (Aggregate constraint) Given a database scheme \mathcal{D} , an aggregate constraint on \mathcal{D} is an expression of the form:

$$\forall x_1, \dots, x_k \left(\phi(x_1, \dots, x_k) \implies \sum_{i=1}^n c_i \cdot \chi_i(X_i) \leq K \right) \quad (6.1)$$

where:

1. c_1, \dots, c_n, K are constants in \mathbb{Q} ;
2. $\phi(x_1, \dots, x_k)$ is a conjunction of relational atoms containing the variables x_1, \dots, x_k and constants;
3. each $\chi_i(X_i)$ is an aggregation function, where X_i is a list of variables and constants, and variables appearing in X_i are a subset of $\{x_1, \dots, x_k\}$.

□

A set of aggregate constraints is denoted as \mathcal{AC} . Given a database D and a set of aggregate constraints \mathcal{AC} , we will use the notation $D \models \mathcal{AC}$ [resp. $D \not\models \mathcal{AC}$] to say that D is consistent [resp. inconsistent] w.r.t. \mathcal{AC} .

Observe that aggregate constraints enable equalities to be expressed as well, since an equality can be viewed as a pair of inequalities. For the sake of brevity, in the following equalities will be written explicitly.

Example 6.3 Constraint 1 defined in Example 6.1 can be expressed as follows:

$$\forall x, y, s, t, v \text{ CashBudget}(y, x, s, t, v) \implies \chi_1(x, y, \text{'det'}) - \chi_1(x, y, \text{'aggr'}) = 0 \quad \square$$

For the sake of simplicity, in the following we will use a shorter notation for denoting aggregate constraints, where universal quantification is implied and variables in ϕ which do not occur in any aggregation function are replaced with the symbol ' $_$ '. For instance, the constraint of Example 6.3 can be written as:

$$\text{CashBudget}(y, x, _ , _ , _) \implies \chi_1(x, y, \text{'det'}) - \chi_1(x, y, \text{'aggr'}) = 0$$

Example 6.4 Constraints 2 and 3 of Example 6.1 can be expressed as follows:

Constraint 2:

$$\text{CashBudget}(x, _ , _ , _ , _) \implies \chi_2(x, \text{'net cash inflow'}) - (\chi_2(x, \text{'total cash receipts'}) - \chi_2(x, \text{'total disbursements'})) = 0$$

Constraint 3:

$$\text{CashBudget}(x, _ , _ , _ , _) \implies \chi_2(x, \text{'ending cash balance'}) - (\chi_2(x, \text{'beginning cash'}) + \chi_2(x, \text{'net cash inflow'})) = 0$$

Consider the database scheme consisting of relation *CashBudget* and relation *Sales*(*Product*, *Year*, *Income*), containing pieces of information on annual product sales. The following aggregate constraint says that, for each year, the value of *cash sales* in *CashBudget* must be equal to the total incomes obtained from relation *Sales*:

$$\text{CashBudget}(x, _ , _ , _ , _) \wedge \text{Sales}(_ , x, _) \implies \chi_2(x, \text{'cash sales'}) - \chi_3(x) = 0$$

where $\chi_3(x)$ is the aggregation function returning the total income due to products sales in year x :

```

 $\chi_3(x) = \text{SELECT sum(Income)}$ 
          FROM Sales
          WHERE Year = x

```

□

6.4 Repairs

Updates at attribute-level will be used in the following as the basic primitives for repairing data violating aggregate constraints. Given a relation scheme P in the database scheme \mathcal{D} , let $\mathcal{M}_P = \{A_1, \dots, A_k\}$ be the subset of $\mathcal{M}_{\mathcal{D}}$ containing all the attributes in P belonging to $\mathcal{M}_{\mathcal{D}}$.

Definition 6.2 (Atomic update) Let $t = P(a_1, \dots, a_n)$ be a tuple on the relation scheme $P(A_1, \dots, A_n)$. An *atomic update* on t is a triplet $\langle t, A_i, a'_i \rangle$, where $A_i \in \mathcal{M}_P$ and a'_i is a value in $DOM(A_i)$ and $a'_i \neq a_i$. \square

Update $u = \langle t, A_i, a'_i \rangle$ replaces $t[A_i]$ with a'_i , thus yielding the tuple $u(t) = P(a_1, \dots, a_{i-1}, a'_i, a_{i+1}, \dots, a_n)$.

Observe that atomic updates work on the set \mathcal{M}_P of measure attributes, as our framework is based on the assumption that data inconsistency is due to errors in the acquisition phase (as in the case of digitization of paper documents) or in the measurement phase (as in the case of sensor readings). Therefore our approach will only consider repairs aiming at re-constructing the correct measures.

Example 6.5 Update $u = \langle t, \text{Value}, 130 \rangle$ issued on the following tuple:

$$t = \text{CashBudget}(2003, \text{'Receipts'}, \text{'cash sales'}, \text{'det'}, 100)$$

returns the tuple:

$$u(t) = \text{CashBudget}(2003, \text{'Receipts'}, \text{'cash sales'}, \text{'det'}, 130).$$

\square

Given an update u , we denote the pair $\langle \text{tuple}, \text{attribute} \rangle$ updated by u as $\lambda(u)$. That is, if $u = \langle t, A_i, a \rangle$ then $\lambda(u) = \langle t, A_i \rangle$.

Definition 6.3 (Consistent database update) Let D be a database and $U = \{u_1, \dots, u_n\}$ be a set of atomic updates on tuples of D . The set U is said to be a *consistent database update* iff $\forall j, k \in [1..n]$ if $j \neq k$ then $\lambda(u_j) \neq \lambda(u_k)$. \square

Informally, a set of atomic updates U is a consistent database update iff for each pair of updates $u_1, u_2 \in U$, u_1 and u_2 do not work on the same tuples, or they change different attributes of the same tuple.

The set of pairs $\langle \text{tuple}, \text{attribute} \rangle$ updated by a consistent database update U will be denoted as

$$\lambda(U) = \bigcup_{u_i \in U} \{\lambda(u_i)\}$$

Given a database D , a tuple t in D , and a consistent database update U , we denote the tuple obtained by applying the atomic updates in U of the form $\langle t, x, y \rangle$ on t as $U(t)$. Moreover, we denote the database resulting from applying all the atomic updates in U on D as $U(D)$.

Definition 6.4 (Repair) Let \mathcal{D} be a database scheme, \mathcal{AC} a set of aggregate constraints on \mathcal{D} , and D an instance of \mathcal{D} such that $D \not\models \mathcal{AC}$. A *repair* ρ for D is a consistent database update such that $\rho(D) \models \mathcal{AC}$. \square

Example 6.6 A repair ρ for *CashBudget* w.r.t. constraints 1), 2) and 3) consists in decreasing attribute *Value* in the tuple:

$$t = \text{CashBudget}(2003, \text{'Receipts'}, \text{'total cash receipts'}, \text{'aggr'}, 250)$$

down to 220; that is, $\rho = \{ \langle t, \text{Value}, 220 \rangle \}$. \square

Example 6.7 Let \mathcal{D} be the database scheme consisting of relation schemes $P_1(K_1, A, B, C)$, $P_2(K_2, D, E, F)$, where K_1 and K_2 are key attributes for P_1 and P_2 , respectively, and $\mathcal{M}_{\mathcal{D}} = \{A, B, C, D, E, F\}$. Let D be the database instance of \mathcal{D} consisting of the following relations (the term t_x denotes the tuple where the key attribute is equal to x):

	K_1	A	B	C
t_{a_1}	a ₁	1	2	3
t_{a_2}	a ₂	1	2	87

P_1

	K_2	D	E	F
t_{b_1}	b ₁	1	4	3
t_{b_2}	b ₂	2	6	6

P_2

Let \mathcal{AC} be the singleton consisting of the following aggregate constraint:

$$\forall x_1, x_2, x_3, x_4, x_5 \ (P_1(x_1, x_2, x_3, 3) \wedge P_2(x_4, x_2, 4, x_5) \implies \chi(x_3) \leq 2)$$

where:

```

χ(x3) = SELECT sum(F)
        FROM   P2
        WHERE  x3 < E

```

Consider the following set of atomic updates on D :

$$\rho = \{ \langle t_{b_1}, D, 3 \rangle, \langle t_{b_1}, E, 1 \rangle, \langle t_{b_2}, D, 1 \rangle, \langle t_{b_2}, E, 4 \rangle, \langle t_{b_2}, F, 1 \rangle \}.$$

It is easy to see that D is not consistent w.r.t. to \mathcal{AC} , and that ρ is a repair for D w.r.t. \mathcal{AC} , as relations P_1 , P_2 in $\rho(D)$ are as follows (circles circumscribe attribute values updated by ρ):

	K_1	A	B	C
$\rho(t_{a_1})$	a ₁	1	2	3
$\rho(t_{a_2})$	a ₂	1	2	87

P_1

	K_2	D	E	F
$\rho(t_{b_1})$	b ₁	(3)	(1)	3
$\rho(t_{b_2})$	b ₂	(1)	(4)	(1)

P_2

\square

6.4.1 Reparability

We now characterize the complexity of the repair-existence problem. All the complexity results in the chapter refer to data-complexity, that is the size of the constraints is assumed to be bounded by a constant.

The following lemma is a preliminary result which states that potential repairs for an inconsistent database can be found among set of updates whose size is polynomially bounded by the size of the original database.

Lemma 6.1 Let \mathcal{D} be a database scheme, \mathcal{AC} a set of aggregate constraints on \mathcal{D} , and D an instance of \mathcal{D} such that $D \not\models \mathcal{AC}$. If there is a repair ρ for D w.r.t. \mathcal{AC} , then there is a repair ρ' for D w.r.t. \mathcal{AC} such that $\lambda(\rho') \subseteq \lambda(\rho)$ and ρ' has polynomial size w.r.t. D .

Proof. Let ρ be a repair for D w.r.t. \mathcal{AC} , and X_ρ the set of variables containing, for each pair $\langle t, A \rangle \in \lambda(\rho)$, a unique variable $x_{t,A}$ ranging on the domain of A . For instance, in the case described in Example 6.7, $X_\rho = \{x_{t_{b_1},D}, x_{t_{b_1},E}, x_{t_{b_2},D}, x_{t_{b_2},E}, x_{t_{b_2},F}\}$.

The intuition underlying the proof is that the existence of ρ implies the existence of a feasible set of linear inequalities $In(\mathcal{AC})$ defined on the variables of X_ρ such that every solution \hat{x} of $In(\mathcal{AC})$ corresponds to a repair $\rho_{\hat{x}}$ for D w.r.t. \mathcal{AC} . In more detail, we will show that, given a solution \hat{x} of $In(\mathcal{AC})$, and denoting the value of variable $x_{t,A}$ in \hat{x} as $\hat{x}_{t,A}$, the set of atomic updates:

$$\rho_{\hat{x}} = \{ \langle t, A, \hat{x}_{t,A} \rangle \mid \langle t, A \rangle \in \lambda(\rho) \wedge \hat{x}_{t,A} \neq t[A] \}$$

is a repair for D w.r.t. \mathcal{AC} .

Specifically, $In(\mathcal{AC})$ can be shown to have a solution \hat{x} of polynomial size w.r.t. D , so that $\rho_{\hat{x}}$ is a polynomial-size repair for D w.r.t. \mathcal{AC} which updates a subset of the values modified by ρ .

Throughout this proof, for the sake of clarity, we will provide several examples for explaining the different steps of the translation of \mathcal{AC} into $In(\mathcal{AC})$. All these examples will be referred to the case described in Example 6.7.

The rest of this proof is organized as follows. We first introduce the definition of $In(\mathcal{AC})$, by explaining how to translate each constraint in \mathcal{AC} into a set of (in)equalities. Then, we prove that $In(\mathcal{AC})$ has at least a solution of polynomial size w.r.t. D , and that every solution of $In(\mathcal{AC})$ defines a repair for D w.r.t. \mathcal{AC} . Finally, we exploit these properties to prove the statement.

————— BEGINNING OF DEFINITION OF $In(\mathcal{AC})$ —————

The set of inequalities $In(\mathcal{AC})$ is obtained by first translating each aggregate constraint ac in \mathcal{AC} into a set of inequalities $In(ac)$, and then assembling the sets of inequalities corresponding to the different constraints into a unique set of inequalities. We focus our attention on defining the translation of a single aggregate constraint of \mathcal{AC} .

Constraint ac is of the form (6.1), with

$$\phi(x_1, \dots, x_k) = P_1(y_1^1, \dots, y_{k_1}^1) \wedge \dots \wedge P_m(y_1^m, \dots, y_{k_m}^m)$$

where P_1, \dots, P_m are (not necessarily distinct) relation names in \mathcal{D} and, for each $l \in [1..m]$, $j \in [1..k_l]$, the term y_j^l is either a variable in $\{x_1, \dots, x_k\}$ or a constant. In the following, we denote the relation scheme of each P_l occurring in ϕ as $P_l(A_1^l, \dots, A_{k_l}^l)$.

Formula $\phi(x_1, \dots, x_k)$ can be re-written into an equivalent formula $\phi'(x_1, \dots, x_k)$ of the form:

$$\begin{aligned} \exists z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m \left(\Psi(z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m) \wedge \right. \\ \left. \Upsilon(z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m, x_1, \dots, x_k) \right) \end{aligned} \quad (6.2)$$

where:

- $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ are new variable symbols not occurring in ϕ ;
- $\Psi(z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m) = P_1(z_1^1, \dots, z_{k_1}^1) \wedge \dots \wedge P_m(z_1^m, \dots, z_{k_m}^m)$;
- $\Upsilon(z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m, x_1, \dots, x_k)$ is a conjunction of equality atoms of the form $z_j^l = T$, where $l \in [1..m]$, $j \in [1..k_l]$, T is either a constant or a variable in $\{z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m, x_1, \dots, x_k\}$, and each variable $x_i \in \{x_1, \dots, x_k\}$ appears in exactly one equality atom.

For instance, the formula $\phi(x_1, x_2, x_3, x_4, x_5) = P_1(x_1, x_2, x_3, 3) \wedge P_2(x_4, x_2, 4, x_5)$ occurring in the aggregate constraint of our running example is equivalent to the formula:

$$\begin{aligned} \phi'(x_1, x_2, x_3, x_4, x_5) = \exists z_1^1, z_2^1, z_3^1, z_4^1, z_1^2, z_2^2, z_3^2, z_4^2 \left(P_1(z_1^1, z_2^1, z_3^1, z_4^1) \right. \\ \wedge P_2(z_1^2, z_2^2, z_3^2, z_4^2) \wedge z_1^1 = z_2^2 \wedge z_4^1 = 3 \wedge z_3^2 = 4 \\ \wedge z_1^1 = x_1 \wedge z_2^1 = x_2 \wedge z_3^1 = x_3 \wedge z_1^2 = x_4 \\ \left. \wedge z_4^2 = x_5 \right) \end{aligned}$$

For the sake of readability, formulas $\Upsilon(z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m, x_1, \dots, x_k)$ and $\Psi(z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m)$ will be referred to as Υ and Ψ , respectively, thus omitting variables.

Let Θ_{ac} be the set of ground substitutions of variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that, for every $\theta \in \Theta_{ac}$, $\theta(\Psi)$ evaluates to true on $\rho(D)$. For each $\theta \in \Theta_{ac}$ and $l \in [1..m]$, we denote as t^l the tuple in relation P_l in D “singled out” by θ , i.e., t^l is the tuple such that $\rho(t^l) = \theta(P_l(z_1^l, \dots, z_{k_l}^l))$. For instance, in our running example, consider the substitution:

$$\theta' = \{z_1^1/a_2, z_2^1/1, z_3^1/2, z_4^1/87, z_1^2/b_1, z_2^2/3, z_3^2/1, z_4^2/3\}$$

such that $\theta'(\Psi) = P_1(a_2, 1, 2, 87) \wedge P_2(b_1, 3, 1, 3)$. Then, we have $t^1 = P_1(a_2, 1, 2, 87)$ and $t^2 = P_2(b_1, 1, 4, 3)$.

For each $\theta \in \Theta_{ac}$, let $\tilde{\theta}$ be the substitution of variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ with either variables in X_ρ or constants defined as follows:

$$\tilde{\theta}(z_j^l) = \begin{cases} x_{t^l, A_j^l} & \text{if } \langle t^l, A_j^l \rangle \in \lambda(\rho); \\ \theta(z_j^l) & \text{otherwise.} \end{cases}$$

Basically, $\tilde{\theta}$ substitutes the variable z_j^l with the variable x_{t^l, A_j^l} in the case that the value of the attribute A_j^l in the tuple t^l in D has been updated by ρ . Otherwise, $\tilde{\theta}$ substitutes z_j^l with $\theta(z_j^l)$, which is the value of attribute A_j^l in both the tuples t^l in D and $\rho(t^l)$ in $\rho(D)$. For instance, in our running example, consider the above-introduced substitution θ' such that $\theta'(\Psi) = P_1(a_2, 1, 2, 87) \wedge P_2(b_1, 3, 1, 3)$. Then, we have that $\tilde{\theta}'(z_4^2) = \theta'(z_4^2) = 3$ and $\tilde{\theta}'(z_3^2) = x_{t_{b_1}, E}$.

The set of inequalities $In(ac)$ corresponding to a single aggregate constraint ac is obtained by assembling different sets of inequalities, which are generated considering the substitutions in Θ_{ac} separately. Specifically, for each $\theta \in \Theta_{ac}$, we define a set of inequalities $In(ac, \theta)$ according to the following cases:

CASE 1:

There is no substitution $\bar{\theta}$ of variables x_1, \dots, x_k with variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that $\theta(\bar{\theta}(\phi'))$ (i.e. $\theta(\Psi) \wedge \theta(\bar{\theta}(\mathcal{T}))$) evaluates to true on $\rho(D)$:

In our running example, this happens, for instance, if we consider the substitution θ' defined above.

Initially, $In(ac, \theta)$ is set to \emptyset . Then, for each equality atom in \mathcal{T} of the form $z_j^l = T$, if the following three conditions are true:

- T is either a variable among $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ or a constant (i.e., T is not a variable in $\{x_1, \dots, x_k\}$);
- $\theta(z_j^l) \neq \theta(T)$;
- at least one among $\tilde{\theta}(z_j^l)$ and $\tilde{\theta}(T)$ is not a constant,

then we add to $In(ac, \theta)$ either the inequality $\tilde{\theta}(z_j^l) > \tilde{\theta}(T)$ (in the case that $\theta(z_j^l) > \theta(T)$) or the inequality $\tilde{\theta}(z_j^l) < \tilde{\theta}(T)$ (in the case that $\theta(z_j^l) < \theta(T)$).

For instance, consider the conjunct $z_3^2 = 4$ occurring in \mathcal{T} in our running example, and the above-defined substitution θ' . As $\tilde{\theta}'(z_3^2) = x_{t_{b_1}, E}$ and $\theta'(z_3^2) = 1 < 4$, we add the inequality $x_{t_{b_1}, E} < 4$ to $In(ac, \theta')$. Analogously, consider the conjunct $z_2^1 = z_2^2$. Since $\theta'(z_2^1) = 1 < \theta'(z_2^2) = 3$ and $\tilde{\theta}'(z_2^1) = 1$ and $\tilde{\theta}'(z_2^2) = x_{t_{b_1}, D}$, we add the inequality $1 < x_{t_{b_1}, D}$ to $In(ac, \theta')$.

Intuitively, in this case, the inequalities of $In(ac, \theta)$ ensure that, if the left-hand side of constraint ac is false w.r.t. $\rho(D)$, then, for every solution \hat{x} of $In(\mathcal{AC})$, it is false w.r.t. $\rho_{\hat{x}}(D)$ too (in a sense that will be made clearer below). Specifically, let t^1, \dots, t^m be the tuples in D singled out by θ , and, for every solution \hat{x} of $In(\mathcal{AC})$, let $\theta_{\rho_{\hat{x}}}$ be the substitution of variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ with constants such that $\theta_{\rho_{\hat{x}}}(\Psi) = \rho_{\hat{x}}(t^1) \wedge \dots \wedge \rho_{\hat{x}}(t^m)$. Then, the inequalities of $In(ac, \theta)$ ensure that there is no substitution $\bar{\theta}_{\rho_{\hat{x}}}$ of variables x_1, \dots, x_k with variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\phi'))$ evaluates to true.

Observe that the number of (in)equalities added to $In(ac, \theta)$ in this case is bounded by $|\mathcal{Y}|$, that is the number of equality atoms occurring in \mathcal{Y} .

CASE 2 :

There is a substitution $\bar{\theta}$ of variables x_1, \dots, x_k with variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that $\theta(\bar{\theta}(\phi'))$ evaluates to true on $\rho(D)$:¹

In our running example, this happens, for instance, if we consider the substitution:

$$\theta'' = \{z_1^1/a_1, z_2^1/1, z_3^1/2, z_4^1/3, z_1^2/b_2, z_2^2/1, z_3^2/4, z_4^2/1\}$$

such that $\theta''(\Psi) = P_1(a_1, 1, 2, 3) \wedge P_2(b_2, 1, 4, 1)$. In this scenario, we have that: $\bar{\theta}'' = \{x_1/z_1^1, x_2/z_2^1, x_3/z_3^1, x_4/z_4^1, x_5/z_4^2\}$.

Basically, in this case, $In(ac, \theta)$ consists of inequalities translating the fact that the right-hand side of ac holds on every database $\rho_{\hat{x}}(D)$ (where \hat{x} corresponds to a solution of $In(\mathcal{AC})$) whenever the variables occurring as argument of every χ_i have been substituted with the values which they are assigned by θ .

The right-hand side of ac is an inequality of the form

$$\sum_{i=1}^n c_i \cdot \chi_i(X_i) \leq K. \quad (6.3)$$

For each $i \in [1..n]$, let P_{χ_i} and α_i be the relation name and the boolean formula occurring in the FROM and WHERE clauses of aggregate function χ_i , respectively. We denote as e_i the attribute expression occurring as argument of the SUM operator in the SELECT clause of χ_i . Without loss of generality, we assume that every α_i is in disjunctive normal form and e_i is either an attribute or a constant.

For every $i \in [1..n]$, we denote as \mathcal{T}_i (resp., \mathcal{F}_i) the set of tuples belonging to relation P_{χ_i} in D such that, for each $t \in \mathcal{T}_i$, the corresponding tuple $\rho(t)$ in $\rho(D)$ satisfies (resp., does not satisfy) $\theta(\bar{\theta}(\alpha_i))$ (see Section 6.2 for the notion of tuple satisfying a boolean formula). For instance, consider

¹ Observe that, as each variable x_i appears in exactly one equality atom in \mathcal{Y} , substitution $\bar{\theta}$ is unique.

our running example (where $n = 1$) and the substitution θ'' . Then, it holds that $\mathcal{T}_1 = \{P_2(b_2, 2, 6, 6)\}$ and $\mathcal{F}_1 = \{P_2(b_1, 1, 4, 3)\}$.

$In(ac, \theta)$ is initially set to \emptyset . Then, we augment $In(ac, \theta)$ with the following sets of (in)equalities:

a. first, we add to $In(ac, \theta)$ the inequality:

$$\sum_{i=1}^n c_i \cdot \sum_{\substack{t \in \mathcal{T}_i \wedge \\ \langle t, e_i \rangle \in \lambda(\rho)}} x_{t, e_i} \leq K', \quad (6.4)$$

where K' is K minus the sum of the addends on the left-hand side of (6.3) which are attribute expressions consisting of either constants or attribute values not changed by ρ , i.e.,²

$$K' = K - \sum_{i=1}^n c_i \cdot \sum_{\substack{t \in \mathcal{T}_i \wedge \\ \langle t, e_i \rangle \notin \lambda(\rho)}} t[e_i] \quad (6.5)$$

For instance, if we consider our running example and substitution θ'' , it is easy to see that the inequality added to $In(ac, \theta'')$ is $x_{t_{b_2}, F} \leq 2$.

b. for each $i \in [1..n]$ and each tuple $t \in \mathcal{T}_i$ we proceed as follows. From definition of \mathcal{T}_i , we have that there is at least one disjunct β in α_i such that $\rho(t)$ satisfies $\theta(\bar{\theta}(\beta))$. For each β such that $\rho(t)$ satisfies $\theta(\bar{\theta}(\beta))$ we proceed as follows. Every conjunct γ in β is of the form $w_1 \diamond w_2$, where \diamond is a comparison operator, and w_1, w_2 are either variables in x_1, \dots, x_k , constants, or attribute names. Before defining the inequalities that must be added to $In(ac, \theta)$ for the pair i, t under consideration, we introduce the following functions defined on the set of variables, constants and attribute names occurring in β :

$$\mu(w) = \begin{cases} \rho(t)[w] & \text{if } w \text{ is an attribute name;} \\ \theta(\bar{\theta}(w)) & \text{if } w \text{ is a variable in } \{x_1, \dots, x_k\}; \\ w & \text{if } w \text{ is a constant;} \end{cases}$$

$$\tilde{\mu}(w) = \begin{cases} x_{t, w} & \text{if } w \text{ is an attribute name and } \langle t, w \rangle \in \lambda(\rho); \\ \rho(t)[w] & \text{if } w \text{ is an attribute name and } \langle t, w \rangle \notin \lambda(\rho); \\ \tilde{\theta}(\bar{\theta}(w)) & \text{if } w \text{ is a variable in } \{x_1, \dots, x_k\}; \\ w & \text{if } w \text{ is a constant.} \end{cases}$$

² With a little abuse of notation, we assume that, if e_i is a constant, then $t[e_i] = e_i$. Moreover, observe that if e_i is a constant, then $\langle t, e_i \rangle \notin \lambda(\rho)$.

For instance, if we consider the tuple $t_{b_2} = P_2(b_2, 2, 6, 6)$ such that $\rho(t_{b_2}) = P_2(b_2, 1, 4, 1)$ in our running example, and substitution θ'' , we have that $\mu(E) = \rho(t_{b_2})[E] = 4$, and $\tilde{\mu}(E) = x_{t_{b_2}, E}$, and $\mu(x_3) = \theta''(\overline{\theta''}(x_3)) = 2$ and $\tilde{\mu}(x_3) = \overline{\theta''}(\overline{\theta''}(x_3)) = 2$.

For each conjunct $w_1 \diamond w_2$ in β such that either $\mu(w_1)$ or $\mu(w_2)$ is a variable, we consider the following cases:

- if \diamond is equal to ' \neq ', then either the inequality $\tilde{\mu}(w_1) < \tilde{\mu}(w_2)$ (in the case that $\mu(w_1) < \mu(w_2)$) or the inequality $\tilde{\mu}(w_1) > \tilde{\mu}(w_2)$ (in the case that $\mu(w_1) > \mu(w_2)$) is added to $In(ac, \theta)$;
- if \diamond is different from ' \neq ', then the (in)equality $\tilde{\mu}(w_1) \diamond \tilde{\mu}(w_2)$ is added to $In(ac, \theta)$.

In our running example, β consists of the only conjunct $x_3 < E$. Hence, if we consider the tuple $t_{b_2} \in \mathcal{T}$ such that $\rho(t_{b_2}) = P_2(b_2, 1, 4, 1)$ and substitution θ'' , since $\mu(x_3) = 2 < \mu(E) = 4$, the inequality $2 < x_{t_{b_2}, E}$ (corresponding to $\tilde{\mu}(x_3) < \tilde{\mu}(E)$) is added to $In(ac, \theta'')$.

Observe that the overall number of (in)equalities added to $In(ac, \theta)$ in this case is bounded by $\sum_{i=1}^n |\mathcal{I}_i| \cdot |\alpha_i|$, where $|\alpha_i|$ denotes the number of comparison atoms occurring in α_i .

- c. for each $i \in [1..n]$ and each tuple $t \in \mathcal{F}_i$, we proceed as follows. Let μ and $\tilde{\mu}$ be functions on the set of variables, constants and attribute names occurring in β defined as in the above case. From definition of \mathcal{F}_i it follows that every disjunct β_j in α_i contains at least a conjunct γ_j such that $\rho(t)$ does not satisfy $\theta(\overline{\theta}(\gamma_j))$. Specifically, conjunct γ_j is of the form $w_1 \diamond w_2$, where \diamond is a comparison operator, and w_1, w_2 are either variables in x_1, \dots, x_k , constants, or attribute names. For each disjunct β_j in α_i , we add to $In(ac, \theta)$ an inequality defined as follows:
- if \diamond is ' $=$ ', then either the inequality $\tilde{\mu}(w_1) < \tilde{\mu}(w_2)$ or $\tilde{\mu}(w_1) > \tilde{\mu}(w_2)$ is added to $In(ac, \theta)$, in the case that either $\mu(w_1) < \mu(w_2)$ or $\mu(w_1) > \mu(w_2)$, respectively;
 - otherwise, the inequality $\tilde{\mu}(w_1) \not\phi \tilde{\mu}(w_2)$ is added to $In(ac, \theta)$, where $\not\phi$ is the comparison operator "opposite" to \diamond (for instance, if \diamond is ' \leq ', then $\not\phi$ is ' $>$ ').

For instance, if we consider, in our running example, the tuple $t_{b_1} \in \mathcal{F}$ such that $\rho(t_{b_1}) = P_1(b_1, 3, 1, 3)$, and the conjunct $x_3 < E$ occurring in the WHERE clause of χ , since $\mu(x_3) = 2 > \mu(E) = 1$, the inequality $2 \geq x_{t_{b_1}, E}$ (corresponding to $\tilde{\mu}(x_3) \geq \tilde{\mu}(E)$) is added to $In(ac, \theta'')$.

Observe that the overall number of (in)equalities added to $In(ac, \theta)$ in this case is bounded by $\sum_{i=1}^n |\mathcal{F}_i| \cdot |\alpha_i|$, where $|\alpha_i|$ denotes the number of comparison atoms occurring in α_i .

The above-defined rules for translating a pair ac, θ into a set of linear inequalities $In(ac, \theta)$ can be used to formally define the set $In(\mathcal{AC})$. Specifically, the set of inequalities $In(\mathcal{AC})$ translating the set of aggregate constraints \mathcal{AC} is defined as follows:

$$In(\mathcal{AC}) = \bigcup_{ac \in \mathcal{AC}} \left(\bigcup_{\theta \in \Theta_{ac}} In(ac, \theta) \right).$$

————— END OF DEFINITION OF $In(\mathcal{AC})$ —————

For the set of inequalities $In(\mathcal{AC})$ introduced above, the properties stated in the following two claims hold.

Claim 6.1 $In(\mathcal{AC})$ has at least one solution of polynomial size w.r.t. D .

Claim 6.2 For each solution \hat{x} of $In(\mathcal{AC})$, the set of updates $\rho_{\hat{x}}$ is a repair for D w.r.t. \mathcal{AC} .

The proofs of these claims are postponed for the sake of readability. We first exploit them to complete the proof of this lemma. Claim 6.1 implies that there is a solution \hat{x} of $In(\mathcal{AC})$ having polynomial size w.r.t. D , and Claim 6.2 implies that the corresponding set of updates $\rho_{\hat{x}}$ is a repair for D w.r.t. \mathcal{AC} . It is easy to see that $\rho_{\hat{x}}$ is of polynomial size w.r.t. D too (as every updated value in $\rho_{\hat{x}}$ is equal to a value in \hat{x}), and that $\rho_{\hat{x}}$ updates a subset of the values updated by ρ (this trivially follows from definition of $\rho_{\hat{x}}$).

PROOF OF CLAIM 6.1.

Obviously, $In(\mathcal{AC})$ has at least one solution, where each variable $x_{t,A} \in X_{\rho}$ is assigned the value v such that $\langle t, A, v \rangle \in \rho$ (it is easy to see that, for each above-described translation rule, every (in)equality added to $In(\mathcal{AC})$ is satisfied by replacing each variable $x_{t,A}$ with $\rho(t)[A]$, i.e. the value assigned by ρ to attribute A in tuple t).

For each aggregate constraint $ac \in \mathcal{AC}$ and each $\theta \in \Theta_{ac}$, every constant occurring in the inequalities of $In(ac, \theta)$ is either a constant in ac or an attribute value in D . Hence, since $|X_{\rho}|$ is bounded by the size of D , the size of each inequality is linearly bounded by the size of D . Moreover, the number of inequalities which $In(ac, \theta)$ consists of is bounded by $|\mathcal{Y}| + 1 + \sum_{i=1}^n |P_i| \cdot |\alpha_i|$. In fact, if CASE 1 holds, $In(ac, \theta)$ is such that $|In(ac, \theta)| \leq |\mathcal{Y}|$ (see description of CASE 1). Otherwise, $In(ac, \theta)$ consists of both inequality (6.4) (CASE 2.a) and the inequalities generated as described in CASE 2.b and CASE 2.c, thus $|In(ac, \theta)| \leq 1 + \sum_{i=1}^n |\mathcal{T}_i| \cdot |\alpha_i| + \sum_{i=1}^n |\mathcal{F}_i| \cdot |\alpha_i| = 1 + \sum_{i=1}^n |P_i| \cdot |\alpha_i|$.

Furthermore, for each $ac \in \mathcal{AC}$, it holds that $|\Theta_{ac}| = \prod_{i=1}^k |P_i|$. Hence, the number of (in)equalities in $In(ac)$ is bounded by $(|\mathcal{Y}| + 1 + \sum_{i=1}^n |P_i| \cdot |\alpha_i|) \cdot \prod_{i=1}^k |P_i|$.

Since the size of each inequality in $In(ac)$ is linearly bounded by the size of D (as explained above), the latter implies that the size of $In(ac)$ is polynomially bounded by the size of D . Finally, since the number of constraints in \mathcal{AC} is independent from the size of D , from the latter it follows that the size of $In(\mathcal{AC})$ is polynomially bounded by the size of D too.

Since $In(\mathcal{AC})$ has at least one solution, there is a solution of $In(\mathcal{AC})$ of polynomial size w.r.t. the size of $In(\mathcal{AC})$ [18, 67]. Hence, as the size of $In(\mathcal{AC})$ is polynomially bounded by the size of D , it holds that $In(\mathcal{AC})$ has at least one solution of polynomial size w.r.t. D . \square

PROOF OF CLAIM 6.2.

Let \hat{x} be a solution of $In(\mathcal{AC})$. We now prove that $\rho_{\hat{x}}$ is a repair for D w.r.t. \mathcal{AC} . Assume by contradiction that there is a solution \hat{x} of $In(\mathcal{AC})$ such that $\rho_{\hat{x}}$ is not a repair for D w.r.t. \mathcal{AC} . This means that there is a constraint ac in \mathcal{AC} such that $\rho_{\hat{x}}(D) \not\models ac$. Constraint ac can be re-written in the form:

$$\forall x_1, \dots, x_k \left(\phi'(x_1, \dots, x_k) \implies \sum_{i=1}^n c_i \cdot \chi_i(X_i) \leq K \right)$$

where ϕ' is of the form (6.2). The fact that $\rho_{\hat{x}}(D) \not\models ac$ implies that there is a ground substitution $\theta_{\rho_{\hat{x}}}$ of variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ and a substitution $\bar{\theta}_{\rho_{\hat{x}}}$ of variables x_1, \dots, x_k with variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that:

- (a) $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\phi'))$ is true on $\rho_{\hat{x}}(D)$, and
- (b) $\sum_{i=1}^n c_i \cdot \chi_i(\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(X_i))) > K$, where every aggregate function χ_i is evaluated on $\rho_{\hat{x}}(D)$.

Let θ be the ground substitution of variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that for each $l \in [1..m]$, being t the tuple in D such that $\rho_{\hat{x}}(t) = \theta_{\rho_{\hat{x}}}(P_l(z_1^l, \dots, z_{k_l}^l))$, it holds that $\rho(t) = \theta(P_l(z_1^l, \dots, z_{k_l}^l))$. Basically, $\theta_{\rho_{\hat{x}}}$ and θ are ground substitutions of variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that, for each $l \in [1..m]$, the tuples $t' = \theta_{\rho_{\hat{x}}}(P_l(z_1^l, \dots, z_{k_l}^l))$ in $\rho_{\hat{x}}(D)$ and the tuple $t'' = \theta(P_l(z_1^l, \dots, z_{k_l}^l))$ in $\rho(D)$ result from updating the same tuple t in D , i.e., $t' = \rho_{\hat{x}}(t)$ and $t'' = \rho(t)$.

We now prove that there is a substitution $\bar{\theta}$ of variables x_1, \dots, x_k with variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ such that $\theta(\bar{\theta}(\phi'))$ is true on $\rho(D)$. Let $\bar{\theta} = \{x_i/z_j^l \mid \text{the conjunct } x_i = z_j^l \text{ appears in } \Upsilon\}$. First, observe that $\bar{\theta}$ is a substitution of variables in $\{x_1, \dots, x_k\}$ with variables $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ since, for each $i \in [1..k]$, variable x_i appears in exactly one conjunct in Υ (thus, $\bar{\theta}$ contains no two distinct pairs $x_i/z_{j_1}^{l_1}, x_i/z_{j_2}^{l_2}$). We now show that $\theta(\bar{\theta}(\phi'))$ is true on $\rho(D)$. We accomplish this reasoning by contradiction, that is we assume that $\theta(\bar{\theta}(\phi'))$ is false on $\rho(D)$. This implies that

there is a conjunct $z_j^l = T$ in Υ (where T is a variable among $z_1^1, \dots, z_{k_1}^1, \dots, z_1^m, \dots, z_{k_m}^m$ or a constant) such that $\theta(z_j^l) \neq \theta(T)$. If both $\tilde{\theta}(z_j^l)$ and $\tilde{\theta}(T)$ are constants, then $\theta_{\rho_{\hat{x}}}(z_j^l) \neq \theta_{\rho_{\hat{x}}}(T)$ holds too (this trivially follows from the above-explained fact that attribute values which are not updated by ρ are not updated by $\rho_{\hat{x}}$), which contradicts that $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\phi'))$ is true on $\rho_{\hat{x}}(D)$. Otherwise, if at least one among $\tilde{\theta}(z_j^l)$ and $\tilde{\theta}(T)$ is not a constant, either the inequality $\tilde{\theta}(z_j^l) < \tilde{\theta}(T)$ or the inequality $\tilde{\theta}(z_j^l) > \tilde{\theta}(T)$ is in $In(\mathcal{AC})$ (see CASE 1). Since \hat{x} is a solution of $In(\mathcal{AC})$, inequality $\tilde{\theta}(z_j^l) < \tilde{\theta}(T)$ (resp., inequality $\tilde{\theta}(z_j^l) > \tilde{\theta}(T)$) implies that $\theta_{\rho_{\hat{x}}}(z_j^l) < \theta_{\rho_{\hat{x}}}(T)$ (resp., $\theta_{\rho_{\hat{x}}}(z_j^l) > \theta_{\rho_{\hat{x}}}(T)$), thus contradicting that $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\phi'))$ is true on $\rho_{\hat{x}}(D)$. This completes the proof of the existence of $\bar{\theta}$.

The existence of $\bar{\theta}$ implies that, when the pair ac, θ is considered to define the inequalities of $In(ac, \theta)$, CASE 2 holds. Observe that this implies that the inequality $\sum_{i=1}^n c_i \cdot \sum_{t \in \mathcal{T}_i \wedge \langle t, e_i \rangle \in \lambda(\rho)} x_{t, e_i} \leq K'$ is in $In(ac, \theta)$ (see CASE 2.a).

We first prove that, for each $i \in [1..n]$ and for each tuple t belonging to P_{X_i} in D , if $\rho(t)$ satisfies $\theta(\bar{\theta}(\alpha_i))$ then $\rho_{\hat{x}}(t)$ satisfies $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\alpha_i))$, and vice versa.

Left-to-right implication: Reasoning by contradiction, assume that $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\alpha_i))$. The fact that $\rho(t)$ satisfies $\theta(\bar{\theta}(\alpha_i))$ implies that there is at least a disjunct β in α_i such that $\rho(t)$ satisfies $\theta(\bar{\theta}(\beta))$. Since $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\alpha_i))$, it must be the case that $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\beta))$. That is, there is a conjunct $w_1 \diamond w_2$ in β such that $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 \diamond w_2))$. We consider two cases separately.

- If both $\tilde{\mu}(w_1)$ and $\tilde{\mu}(w_2)$ are constants, then we are in the case that neither w_1 nor w_2 refer to database values updated by ρ . Thus, since $\rho_{\hat{x}}$ updates a subset of the values modified by ρ , it must hold that $\rho_{\hat{x}}(t)$ satisfies $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 \diamond w_2))$, which is a contradiction.
- If at least one among $\tilde{\mu}(w_1)$ and $\tilde{\mu}(w_2)$ is a variable, if \diamond is equal to \neq , then either the inequality $\tilde{\mu}(w_1) < \tilde{\mu}(w_2)$ or the inequality $\tilde{\mu}(w_1) > \tilde{\mu}(w_2)$ has been added to $In(ac, \theta)$. Since \hat{x} satisfies inequalities of $In(ac, \theta)$, it must be the case that $\rho_{\hat{x}}(t)$ satisfies $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 < w_2))$ (resp., $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 > w_2))$) and then it satisfies $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 \neq w_2))$ too, which is a contradiction. An analogous reasoning can be exploited in the case that \diamond is different from ' \neq '.

Right-to-left implication: Reasoning by contradiction, assume that $\rho(t)$ does not satisfy $\theta(\bar{\theta}(\alpha_i))$. In this case, for every disjunct β_j in α_i , there is a conjunct $\gamma_j = w_1 \diamond w_2$ such that $\rho(t)$ does not satisfy $\theta(\bar{\theta}(w_1 \diamond w_2))$. We first show that this implies that also $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 \diamond w_2))$. We consider separately the following two cases.

- If both $\tilde{\mu}(w_1)$ and $\tilde{\mu}(w_2)$ are constants, then neither w_1 nor w_2 refer to database values updated by ρ . Thus, since $\rho_{\hat{x}}$ updates a subset of the values modified by ρ , it must hold that $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 \diamond w_2))$, which implies that $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\beta_j))$.
- If at least one among $\tilde{\mu}(w_1)$ and $\tilde{\mu}(w_2)$ is a variable, if \diamond is '=', then either the inequality $\tilde{\mu}(w_1) < \tilde{\mu}(w_2)$ or the inequality $\tilde{\mu}(w_1) > \tilde{\mu}(w_2)$ has been added to $In(ac, \theta)$. Since \hat{x} satisfies inequalities of $In(ac, \theta)$, it must be the case that $\rho_{\hat{x}}(t)$ satisfies $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 < w_2))$ (resp., $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 > w_2))$) and then it does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(w_1 = w_2))$. An analogous reasoning can be exploited in the case that \diamond is different from '='.

This reasoning can be applied to every disjunct β_j in α_i , that is for every disjunct β_j in α_i there is a conjunct γ_j such that $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\gamma_j))$. This implies that $\rho_{\hat{x}}(t)$ does not satisfy $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\alpha_i))$, which contradicts the hypothesis.

We now show that the above-proved equivalence

$$“\forall i \in [1..n], \forall t \in P_{\chi_i} \text{ in } D \ (\rho(t) \text{ satisfies } \theta(\bar{\theta}(\alpha_i)) \Leftrightarrow \rho_{\hat{x}}(t) \text{ satisfies } \theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\alpha_i)))”$$

together with the fact that $\sum_{i=1}^n c_i \cdot \sum_{t \in \mathcal{T}_i \wedge \langle t, e_i \rangle \in \lambda(\rho)} x_{t, e_i} \leq K'$ is in $In(\mathcal{AC})$, imply that $\sum_{i=1}^n c_i \cdot \chi_i(\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(X_i))) \leq K$ holds on $\rho_{\hat{x}}(D)$. We accomplish this reasoning by contradiction, that is we assume that $\sum_{i=1}^n c_i \cdot \chi_i(\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(X_i))) > K$ holds on $\rho_{\hat{x}}(D)$.

For every $i \in [1..n]$, we denote as $\hat{\mathcal{T}}_i$ (resp., $\hat{\mathcal{F}}_i$) the set of tuples belonging to relation P_{χ_i} in D such that, for each $t \in \hat{\mathcal{T}}_i$, the corresponding tuple $\rho_{\hat{x}}(t)$ in $\rho_{\hat{x}}(D)$ satisfies (resp., does not satisfy) $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\alpha_i))$.

From definition of aggregate function, it holds that:

$$\sum_{i=1}^n c_i \cdot \chi_i(\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(X_i))) = \sum_{i=1}^n c_i \cdot \sum_{t \in \hat{\mathcal{T}}_i} \rho_{\hat{x}}(t)[e_i]$$

Due to contradiction hypothesis, the latter implies that:

$$\sum_{i=1}^n c_i \cdot \sum_{t \in \hat{\mathcal{T}}_i} \rho_{\hat{x}}(t)[e_i] > K$$

Since for each $i \in [1..n]$ and for each tuple t belonging to P_{χ_i} in D , if $\rho(t)$ satisfies $\theta(\bar{\theta}(\alpha_i))$ then $\rho_{\hat{x}}(t)$ satisfies $\theta_{\rho_{\hat{x}}}(\bar{\theta}_{\rho_{\hat{x}}}(\alpha_i))$, and vice versa, the latter inequality can be re-written as follows, by making the inner summation range on the tuples of \mathcal{T}_i instead of $\hat{\mathcal{T}}_i$:

$$\sum_{i=1}^n c_i \cdot \sum_{t \in \mathcal{T}_i} \rho_{\hat{x}}(t)[e_i] > K$$

which is equivalent to:

$$\sum_{i=1}^n c_i \cdot \sum_{\substack{t \in \mathcal{T}_i \wedge \\ \langle t, e_i \rangle \in \lambda(\rho)}} \rho_{\hat{x}}(t)[e_i] > K - \sum_{i=1}^n c_i \cdot \sum_{\substack{t \in \mathcal{T}_i \wedge \\ \langle t, e_i \rangle \notin \lambda(\rho)}} \rho_{\hat{x}}(t)[e_i]$$

Since ρ and $\rho_{\hat{x}}$ coincide on the database values not updated by ρ , the right-hand side of the above inequality coincides with the constant K' introduced in equation (6.5) - CASE 2.a. Hence, the above inequality implies that:

$$\sum_{i=1}^n c_i \cdot \sum_{\substack{t \in \mathcal{T}_i \wedge \\ \langle t, e_i \rangle \in \lambda(\rho)}} \rho_{\hat{x}}(t)[e_i] > K'$$

As from the definition of $\rho_{\hat{x}}$, for each $i \in [1..n]$ and for each tuple $t \in \mathcal{T}_i$ such that $\langle t, e_i \rangle \in \lambda(\rho)$, it holds that $\rho_{\hat{x}}(t)[e_i] = \hat{x}_{t, e_i}$, the latter inequality implies that:

$$\sum_{i=1}^n c_i \cdot \sum_{\substack{t \in \mathcal{T}_i \wedge \\ \langle t, e_i \rangle \in \lambda(\rho)}} \hat{x}_{t, e_i} > K'$$

which contradicts the hypothesis that \hat{x} is a solution of $In(\mathcal{AC})$, since the inequality $\sum_{i=1}^n c_i \cdot \sum_{\substack{t \in \mathcal{T}_i \wedge \\ \langle t, e_i \rangle \in \lambda(\rho)}} x_{t, e_i} \leq K'$ has been added to $In(\mathcal{AC})$ (see CASE 2.a).

This completes the proof of the property that, for each solution \hat{x} of $In(\mathcal{AC})$, the corresponding set of updates $\rho_{\hat{x}}$ is a repair for D w.r.t. \mathcal{AC} . \square

Theorem 6.1 (Repair existence) Let \mathcal{D} be a database scheme, \mathcal{AC} a set of aggregate constraints on \mathcal{D} , and D an instance of \mathcal{D} such that $D \not\models \mathcal{AC}$. The problem of deciding whether there is a repair for D is *NP*-complete (w.r.t. the size of D).

Proof. Membership. A polynomial size witness for deciding the existence of a repair is a database update U on D : testing whether U is a repair for D means verifying $U(D) \models \mathcal{AC}$, which can be accomplished in polynomial time w.r.t. the size of D and U . If a repair exists for D , then Lemma 6.1 guarantees that a polynomial size repair for D exists too.

Hardness. We show a reduction from CIRCUIT SAT to our problem. Without loss of generality, we consider a boolean circuit C using only NOR gates. The inputs of C will be denoted as x_1, \dots, x_n . The boolean circuit C can be represented by means of the database scheme:

$gate(\underline{IDGate}, norVal, orVal),$
 $gateInput(\underline{IDGate}, \underline{IDIngoing}, Val),$
 $input(\underline{IDInput}, Val).$

Therein:

1. each gate in C corresponds to a tuple in $gate$ (attributes $norVal$ and $orVal$ represent the output of the corresponding NOR gate and its negation, respectively);
2. inputs of C correspond to tuples of $input$: attribute Val in a tuple of $input$ represents the truth assignment to the input $x_{IDInput}$;
3. each tuple in $gateInput$ represents an input of the gate identified by $IDGate$. Specifically, $IDIngoing$ refers to either a gate identifier or an input identifier; attribute Val is a copy of the truth value of the specified ingoing gate or input.

We consider the database instance D where the relations defined above are populated as follows. For each input x_i in C we insert the tuple $input(id(x_i), -1)$ into D , and for each gate g in C we insert the tuple $gate(id(g), -1, -1)$, where function $id(x)$ assigns a unique identifier to its argument (we assume that gate identifiers are distinct from input identifiers, and that the output gate of C is assigned the identifier 0). Moreover, for each edge in C going from g' to the gate g (where g' is either a gate or an input of C), the tuple $gateInput(id(g), id(g'), -1)$ is inserted into D . Assume that $\mathcal{M}_{gate} = \{norVal, orVal\}$, $\mathcal{M}_{gateInput} = \{Val\}$, $\mathcal{M}_{input} = \{Val\}$. In the following, we will define aggregate constraints to force measure attributes of all tuples to be assigned either 1 or 0, representing the truth value *true* and *false*, respectively. The initial assignment (where every measure attribute is set to -1) means that the truth values of inputs and gate outputs is undefined.

Consider the following aggregation functions:

$ \begin{aligned} NORVal(X) &= \text{SELECT Sum}(norVal) \\ &\quad \text{FROM gate} \\ &\quad \text{WHERE (IDGate=X)} \end{aligned} $	$ \begin{aligned} ORVal(X) &= \text{SELECT Sum}(orVal) \\ &\quad \text{FROM gate} \\ &\quad \text{WHERE (IDGate=X)} \end{aligned} $
$ \begin{aligned} IngoingVal(X, Y) &= \text{SELECT Sum}(Val) \\ &\quad \text{FROM gateInput} \\ &\quad \text{WHERE (IDGate=X)} \\ &\quad \quad \text{AND (IDIngoing=Y)} \end{aligned} $	$ \begin{aligned} IngoingSum(X) &= \text{SELECT Sum}(Val) \\ &\quad \text{FROM gateInput} \\ &\quad \text{WHERE (IDGate=X)} \end{aligned} $

$$\begin{aligned}
InputVal(X) &= \text{SELECT Sum}(Val) & ValidInput() &= \text{SELECT Sum}(1) \\
&\text{FROM Input} && \text{FROM input} \\
&\text{WHERE (IDInput=X)} && \text{WHERE (Val} \neq 0) \\
&&& \text{AND (Val} \neq 1) \\
\\
ValidGate() &= \text{SELECT Sum}(1) \\
&\text{FROM gate} \\
&\text{WHERE (orVal} \neq 0 \text{ AND orVal} \neq 1) \\
&\text{OR (norVal} \neq 0 \text{ AND norVal} \neq 1)
\end{aligned}$$

Therein: $NORVal(X)$ and $ORVal(X)$ return the truth value of the gate X and its opposite, respectively; $IngoingVal(X, Y)$ returns, for the gate with identifier X , the truth value of the ingoing gate or input having identifier Y ; $IngoingSum(X)$ returns the sum of the truth values of the inputs of the gate X ; $InputVal(X)$ returns the truth assignment of the input X ; $ValidInput()$ returns 0 iff there is no tuple in relation *input* where attribute *Val* is neither 0 nor 1, otherwise it returns a number greater than 0; likewise, $ValidGate()$ returns 0 iff there is no tuple in relation *gate* where attributes *norVal* or *orVal* are neither 0 nor 1 (otherwise it returns a number greater than 0).

Consider the following aggregate constraints on \mathcal{D} :

1. $ValidInput() + ValidGate() = 0$, which entails that only 0 and 1 can be assigned either to attributes *orVal* and *norVal* in relation *gate*, and to attribute *Val* in relation *input*;
2. $gate(X, -, -) \Rightarrow ORVal(X) + NORVal(X) = 1$, which says that for each tuple representing a NOR gate, the value of *orVal* must be complementary to *norVal*;
3. $gate(X, -, -) \Rightarrow ORVal(X) - IngoingSum(X) \leq 0$, which says that for each tuple representing a NOR gate, the value of *orVal* cannot be greater than the sum of truth assignments of its inputs (i.e. if all inputs are 0, *orVal* must be 0 too);
4. $gateInput(X, Y, -) \Rightarrow IngoingVal(X, Y) - ORVal(X) \leq 0$, which implies that, for each gate g , attribute *orVal* must be 1 if at least one input of g has value 1;
5. $gateInput(X, Y, -) \Rightarrow IngoingVal(X, Y) - NORVal(Y) - InputVal(Y) = 0$, which imposes that the attribute *Val* in each tuple of *gateInput* is the same as the truth value of either the ingoing gate or the ingoing input.

Observe that D does not satisfy these constraints, but every repair of D corresponds to a valid truth assignment of C .

Let \mathcal{AC} be the set of aggregate constraints consisting of constraints 1-5 defined above plus constraint $NORVal(0) = 1$ (which imposes that the truth value of the output gate must be *true*). Therefore, deciding whether there is a truth assignment which evaluates C to *true* is equivalent to asking whether there is a repair ρ for D w.r.t. \mathcal{AC} .

□

Theorem 6.1 states that the repair existence problem is decidable. This result, together with the practical usefulness of the considered class of constraints, makes the complexity analysis of finding consistent answers on inconsistent data interesting. Basically decidability results from the linear nature of the considered constraints. If products between two attributes were allowed as attribute expressions, the repair-existence problem would be undecidable. This can be proved straightforwardly, since this form of non-linear constraints is more expressive than those introduced in [16], where the corresponding repair-existence problem was shown to be undecidable (cfr. Section 5.2.2). However, observe that occurrences of products of the form $A_i \times A_j$ in attribute expressions can lead to undecidability only if both A_i and A_j are *measure attribute*. Otherwise, this case is equivalent to products of the form $c \times A$, which can be expressed in our form of aggregate constraints.

6.4.2 Minimal Repairs

Theorem 6.1 deals with the problem of deciding whether a database D violating a set of aggregate constraints \mathcal{AC} can be repaired. If this is the case, different repairs can be performed on D yielding a new database consistent w.r.t. \mathcal{AC} , although not all of them can be considered “reasonable”. For instance, if a repair exists for D changing only one value in one tuple of D , any repair updating all values in all tuples of D can be reasonably disregarded. To evaluate whether a repair should be considered “relevant” or not, we introduce two different ordering criteria on repairs, corresponding to the comparison operators ‘ \leq_{set} ’ and ‘ \leq_{card} ’. The former compares two repairs by evaluating whether one of the two performs a subset of the updates of the other. That is, given two repairs ρ_1, ρ_2 , we say that ρ_1 precedes ρ_2 ($\rho_1 \leq_{set} \rho_2$) iff $\lambda(\rho_1) \subseteq \lambda(\rho_2)$. The latter ordering criterion states that a repair ρ_1 is preferred w.r.t. a repair ρ_2 ($\rho_1 \leq_{card} \rho_2$) iff $|\lambda(\rho_1)| \leq |\lambda(\rho_2)|$, that is if the number of changes issued by ρ_1 is less than ρ_2 .

Observe that $\rho_1 \leq_{set} \rho_2$ implies $\rho_1 <_{card} \rho_2$, but the vice versa does not hold, as it can be the case that repair ρ_1 changes a set of values $\lambda(\rho_1)$ which is not a subset of $\lambda(\rho_2)$, but whose cardinality is less than that of $\lambda(\rho_2)$.

Example 6.8 Another repair for *CashBudget* is:

$$\rho' = \{\langle t_1, Value, 130 \rangle, \langle t_2, Value, 70 \rangle, \langle t_3, Value, 190 \rangle\},$$

where:

$$t_1 = CashBudget(2003, 'Receipts', 'cash sales', 'det', 100),$$

$$t_2 = CashBudget(2003, 'Disbursements', 'long-term financing', 'det', 40),$$

$$t_3 = CashBudget(2003, 'Disbursements', 'total disbursements', 'aggr', 160).$$

Observe that $\rho <_{card} \rho'$, but not $\rho <_{set} \rho'$ (where ρ is the repair defined in Example 6.6).

□

Definition 6.5 (Minimal repairs) Let \mathcal{D} be a database scheme, \mathcal{AC} a set of aggregate constraints on \mathcal{D} , and D an instance of \mathcal{D} . A repair ρ for D w.r.t. \mathcal{AC} is a *set-minimal* repair [resp. *card-minimal* repair] iff there is no repair ρ' for D w.r.t. \mathcal{AC} such that $\rho' <_{\text{set}} \rho$ [resp. $\rho' <_{\text{card}} \rho$]. \square

Example 6.9 Repair ρ of Example 6.6 is minimal under both the *set-minimal* and the *card-minimal* semantics, whereas ρ' defined in Example 6.8 is minimal only under the *set-minimal* semantics.

Consider the repair $\rho'' = \{\langle t_1, \text{Value}, 110 \rangle, \langle t_2, \text{Value}, 110 \rangle, \langle t_3, \text{Value}, 220 \rangle\}$ where:

$$\begin{aligned} t_1 &= \text{CashBudget}(2003, \text{'Receipts'}, \text{'cash sales'}, \text{'det'}, 100), \\ t_2 &= \text{CashBudget}(2003, \text{'Receipts'}, \text{'receivables'}, \text{'det'}, 120), \\ t_3 &= \text{CashBudget}(2003, \text{'Receipts'}, \text{'total cash receipts'}, \text{'aggr'}, 250). \end{aligned}$$

The strategy adopted by ρ'' can be reasonably disregarded, since the only atomic update on tuple t_3 suffices to make D consistent. In fact, ρ'' is not minimal neither under the *set-minimal* semantics (as $\lambda(\rho) \subset \lambda(\rho'')$) and thus $\rho <_{\text{set}} \rho''$) nor under the *card-minimal* one. \square

Given a database D which is not consistent w.r.t. a set of aggregate constraints \mathcal{AC} , different *set-minimal* repairs (resp. *card-minimal* repairs) can exist on D . In our running example, repair ρ of Example 6.6 is the unique *card-minimal* repair, and both ρ and ρ' are *set-minimal* repairs (where ρ' is the repair defined in Example 6.8). The set of *set-minimal* repairs and *card-minimal* repairs will be denoted, respectively, as ρ_M^{set} and ρ_M^{card} .

Observe that, both ρ_M^{set} and ρ_M^{card} contain subsets of consistent database updated. For a database D and a set of aggregate constraints \mathcal{AC} , the set of repaired database instance according to the *set-minimal* semantics (resp. *card-minimal* repairs) will be denoted as $\mathcal{R}(D, \mathcal{AC}, \text{set})$ (resp. $\mathcal{R}(D, \mathcal{AC}, \text{card})$). That is, $\mathcal{R}(D, \mathcal{AC}, \text{set}) = \{\rho(D) \mid \rho \in \rho_M^{\text{set}}\}$ and $\mathcal{R}(D, \mathcal{AC}, \text{card}) = \{\rho(D) \mid \rho \in \rho_M^{\text{card}}\}$.

Theorem 6.2 (Minimal-repair checking) Let \mathcal{D} be a database scheme, \mathcal{AC} a set of aggregate constraints on \mathcal{D} , and D an instance of \mathcal{D} such that $D \not\models \mathcal{AC}$. Given a repair ρ for D w.r.t. \mathcal{AC} , deciding whether ρ is minimal (under either the *card-minimality* and *set-minimality* semantics) is coNP-complete (w.r.t. the size of D and ρ).

Proof. Membership. A polynomial size witness for the complement of the problem of deciding whether $\rho \in \rho_M^{\text{set}}$ [resp. $\rho \in \rho_M^{\text{card}}$] is a repair ρ' such that $\rho' <_{\text{set}} \rho$ [resp. $\rho' <_{\text{card}} \rho$]. From Lemma 6.1 we have that ρ' can be found among repairs having polynomial size w.r.t. D .

Hardness. We show a reduction of MINIMAL MODEL CHECKING (MMC) [22] to our problem. Consider an instance $\langle f, M \rangle$ of MMC, where f is a propositional

formula and M a model for f . Formula f can be translated into an equivalent boolean circuit C using only NOR gates, and C can be represented as shown in the hardness proof of Theorem 6.1. Therefore, we consider the same database scheme \mathcal{D} and the same set of aggregate constraints \mathcal{AC} on \mathcal{D} as those in the proof of Theorem 6.1. Let D be the instance of \mathcal{D} constructed as follows. For each input x_i in C we insert the tuple $input(id(x_i), 0)$ into D . Then, as for the construction in the hardness proof of Theorem 6.1, for each gate g in C we insert the tuple $gate(id(g), -1, -1)$ into D , and for each edge in C going from g' to the gate g (where g' is either a gate or an input of C), the tuple $gateInput(id(g), id(g'), -1)$ is inserted into D .

Observe that any repair for D must update all measure attributes in D with value -1 . Therefore, given two repairs ρ', ρ'' , it holds that for each $\langle t, A \rangle \in (\lambda(\rho') \triangle \lambda(\rho''))$, t is a tuple of $input$ and $A = Val$.

Obviously, a repair ρ for D exists, consisting of the following updates: 1) attribute Val is assigned 1 in every tuple of $input$ corresponding to an atom in f which is true in M ; 2) attributes $norVal$, $orVal$ in $gate$ and Val in $gateInput$ are updated accordingly to updates described above. Basically, such a constructed repair ρ corresponds to M (we say that a repair corresponds to a model if it assigns 1 to attribute Val in the tuples of $input$ corresponding to the atoms which are true in the model, 0 otherwise).

If M is not a minimal model for f , then there exists a model M' such that $M' \subset M$ (i.e. atoms which are true in M' are a proper subset of atoms which are true in M). Then, the repair ρ' corresponding to M' satisfies $\rho' <_{set} \rho$. Vice versa, if there exists a repair ρ' such that $\rho' <_{set} \rho$, then the model M' corresponding to ρ' is a proper subset of M , thus M is not minimal. This proves that M is a minimal model for f iff ρ is a minimal repair (under set -minimal semantics) for D w.r.t \mathcal{AC} .

Proving hardness under $card$ -minimal semantics can be accomplished as follows. First, a formula f_M is constructed from f by replacing, for each atom $\alpha \notin M$, each occurrence of α in f with the contradiction $(\alpha \wedge \neg\alpha)$. Then, an instance D of \mathcal{D} is constructed corresponding to formula f_M with the same value assignments as before (attribute Val in all the tuples of $input$ are set to 0, and all the other measure attributes are set to -1).

M is a model for both f and f_M , and it is minimal for f iff it is minimum for f_M . In fact, if M is minimal for f there is no subset M' of M which is a model of f . Then, assume that a model M'' for f_M exists, such that $|M''| < |M|$. Then, also $M''' = M'' \cap M$ is a model for f_M , implying that M''' is a model for f , which is a contradiction (as $M''' \subset M$). On the other hand, if M is minimum for f_M then M must be minimal for f . Otherwise, there would exist a model M' for f s.t. $M' \subset M$. However M' is also a model for f_M , which is a contradiction, as $|M'| < |M|$.

Let ρ be the repair of D w.r.t. \mathcal{AC} corresponding to M . If M is not minimum, then there exists M' (with $|M'| < |M|$) which is a model for f_M . Therefore the repair ρ' corresponding to M' satisfies $\rho' <_{card} \rho$. Vice versa, if a repair ρ' for D w.r.t. \mathcal{AC} exists such that $\rho' <_{card} \rho$, then the model M'

corresponding to ρ' is such that $|M'| < |M|$, thus M is not minimum for f_M . This proves that M is a minimal model for f iff there is no repair ρ' for D w.r.t. \mathcal{AC} such that $\rho' <_{card} \rho$. \square

6.4.3 Set-Minimality versus Card-Minimality

Basically, both the *set*-minimal and the *card*-minimal semantics aim at considering “reasonable” repairs which preserve the content of the input database as much as possible. The notion of repair minimality based on the number of performed updates has been discussed in the context of relational data violating “non-numerical” constraints in [42, 9, 65, 66]. Indeed, most of the proposed approaches consider repairs consisting of deletions and insertions of tuples, and preferred repairs are those consisting of minimal sets of insert/delete operations (cfr. Chapter 3 and Chapter 4). In fact, the *set*-minimal semantics is more natural than the *card*-minimal one when no hypothesis can be reasonably formulated to “guess” how data inconsistency occurred, which is the case of previous works on database-repairing. As it will be clear in the following, in the general case, the adoption of the *card*-minimal semantics could make reasonable sets of delete/insert operations to be not considered as candidate repairs, even if they correspond to error configurations which cannot be excluded.

For instance, consider the relation scheme:

Department(*Name*, *Area*, *Employees*, *Category*)

and the relation:

<i>Name</i>	<i>Area</i>	<i>Employees</i>	<i>Category</i>	
D_1	100	24	A	$\longrightarrow t_1$
D_2	100	30	B	$\longrightarrow t_2$
D_3	100	30	B	$\longrightarrow t_3$

where the following functional dependencies are defined:

- $FD_1: Area \rightarrow Employees$ (i.e. departments having the same area must have the same number of employees)
 $FD_2: Employees \rightarrow Category$ (i.e. departments with the same number of employees must be of the same category)

The above-reported relation does not satisfy FD_1 , as the three departments occupy the same area but do not have the same number of employees. Suppose we are using a repairing strategy based on deletions and insertions of tuples. Different repairs can be adopted. For instance, if we suppose that the inconsistency arises as tuple t_1 contains wrong information, *Department* can be repaired by only deleting t_1 . Otherwise, if we assume that t_1 is correct, a possible repair consists of deleting t_2 and t_3 . If the *card*-minimal semantics is

adopted, the latter strategy will be disregarded, as it performs two deletions, whereas the former deletes only one tuple. On the contrary, if the *set*-minimal semantics is adopted, both the two strategies define minimal repairs (as the sets of tuples deleted by each of these strategies are not subsets of one another). In fact, if we do not know how the error occurred, there is no reason to assume that the error configuration corresponding to the second repairing strategy is not possible. Indeed, inconsistency could be due to integrating data coming from different sources, where some sources are not up-to-date. However, there is no good reason to assume that the source which contains the smallest number of tuples is the one that is up to date. See [60] for a survey on inconsistency due to data integration.

Likewise, the *card*-minimal semantics could disregard reasonable repairs also in the case that a repairing strategy based on updating values instead of deleting/inserting whole tuples is adopted³. For instance, if we suppose that the inconsistency arises as the value of attribute *Area* is wrong for either t_1 or both t_2 and t_3 , *Department* can be repaired by replacing the *Area* value for either t_1 or both t_2 and t_3 with a value different from 100. Otherwise, if we assume that the *Area* values for all the tuples are correct, *Department* can be repaired w.r.t. FD_1 by making the *Employees* value of t_1 equal to that of t_2 and t_3 . Indeed this update yields a relation which does not satisfy FD_2 (as $t_1[Category] \neq t_2[Category]$) so that another value update is necessary in order to make it consistent. Under the *card*-minimal semantics the latter strategy is disregarded, as it performs more than one value update, whereas the former changes only the *Area* value of one tuple. On the contrary, under the *set*-minimal semantics both the two strategies define minimal repairs (as the sets of updates issued by each of these strategies are not subsets of one another). As for the case explained above, disregarding the second repairing strategy is arbitrary, if we do not know how the error occurred.

Our framework addresses scenarios where also *card*-minimal semantics can be reasonable. For instance, if we assume that integrity violations are generated while acquiring data by means of an automatic or semi-automatic system (e.g. an OCR digitizing a paper document, a sensor monitoring atmospheric conditions, etc.), focusing on error configurations which can be repaired with the minimum number of updates is well founded. Indeed this corresponds to the case that the acquiring system made the minimum number of errors (e.g. bad symbol-recognition for an OCR, sensor troubles, etc.), which can be considered the most probable event.

In this chapter we discuss the existence of repairs, and their computation under both *card*-minimal and *set*-minimal semantics. The latter has to be preferred when no warranty is given on the accuracy of acquiring tools, and, more generally, when no hypothesis can be formulated on the cause of errors.

³ Value updates cannot be necessarily simulated as a sequence deletion/insertion, as this might not be minimal under set inclusion.

6.5 Consistent Query Answers

In this section we address the problem of extracting reliable information from data violating a given set of aggregate constraints. We consider boolean queries checking whether a given tuple belongs to a database, i.e. queries of the form $Q = P(a_1, \dots, a_n)$, where $P(A_1, \dots, A_n)$ is a relation scheme in \mathcal{D} . We adopt the widely-used notion of consistent query answer introduced in [4] (cfr. Section 2.5).

We recall the notion of *consistent query answer* (cfr. Definition 1.7). Let \mathcal{D} be a database scheme, D be an instance of \mathcal{D} , \mathcal{AC} be a set of aggregate constraints on \mathcal{D} and Q be a boolean ground query over \mathcal{D} . The consistent query answer of Q on D under the *set-minimal* semantics (resp. *card-minimal* semantics) is true iff $q \in \rho(D)$ for each $\rho \in \rho_M^{\text{set}}$ (resp. for each $\rho \in \rho_M^{\text{card}}$).

The consistent query answers of a query Q issued on the database D under the *set-minimal* and *card-minimal* semantics will be denoted as $Q^{\text{set}}(D)$ and $Q^{\text{card}}(D)$, respectively. The following theorems characterize data-complexity of the consistent query answering problem under both the *set-minimal* and *card-minimal* semantics. Before providing the theorems, we prove two lemmas. The former (resp. the latter) lemma ensures that if there is a repair ρ for a database D such that a boolean ground query Q is true (resp. false) in $\rho(D)$, then there is a polynomially bounded repair ρ' for D such that Q is still true (resp. false) in $\rho'(D)$.

Lemma 6.2 Let \mathcal{D} be a database scheme, \mathcal{AC} a set of aggregate constraints on \mathcal{D} , D an instance of \mathcal{D} such that $D \not\models \mathcal{AC}$, and $Q = P(a_1, \dots, a_n)$ a ground boolean query over \mathcal{D} . If there is a repair ρ for D w.r.t. \mathcal{AC} such that $Q \in \rho(D)$, then there is a repair ρ' for D such that (i) $\lambda(\rho') \subseteq \lambda(\rho)$, (ii) $Q \in \rho'(D)$, and (iii) ρ' has polynomial size w.r.t. D and Q .

Proof. Consider the system of inequalities $In(\mathcal{AC})$ obtained from \mathcal{AC} as explained in the proof of Lemma 6.1 (we recall that $In(\mathcal{AC})$ results from translating each aggregate constraint in \mathcal{AC} into a set of inequalities, where each variable $x_{t,A}$ corresponds to a pair $\langle t, A \rangle \in \lambda(\rho)$).

Let τ be the tuple in P which coincides with Q in all the key attributes of P . Observe that the existence of τ is implied by the existence of ρ : since no repair can change key attributes in a relation, the fact that $Q \in \rho(D)$ implies that there is one tuple τ in P such that, for each $A \in \mathcal{K}_P$, $\tau[A] = Q[A]$.

Consider the system of inequalities $In(\mathcal{AC})^+$ obtained from $In(\mathcal{AC})$ by adding, for each $A \in \mathcal{M}_P$ such that $\langle \tau, A \rangle \in \lambda(\rho)$, the equation $x_{\tau,A} = Q[A]$.

Reasoning as in the prof of Claim 6.1 it easy to prove that $In(\mathcal{AC}^+)$ has at least one solution \hat{x} of polynomial size w.r.t. D and Q . Moreover, reasoning as in the prof of Claim 6.2 it is easy to see that, let $\hat{x}_{t,A}$ be the value of variable $x_{t,A}$ in the solution \hat{x} of $In(\mathcal{AC})^+$, then the set of updates

$$\rho_{\hat{x}} = \{ \langle t, A, \hat{x}_{t,A} \rangle \mid \langle t, A \rangle \in \lambda(\rho) \wedge \hat{x}_{t,A} \neq t[A] \}$$

is a repair for D w.r.t. \mathcal{AC} such that $Q \in \rho_{\hat{x}}(D)$ (the fact that $Q \in \rho_{\hat{x}}(D)$ is entailed by the set of equations added to $In(\mathcal{AC})$ in order to obtain $In(\mathcal{AC})^+$). It is easy to see that $\rho_{\hat{x}}$ is of polynomial size w.r.t. D and Q too, and that $\rho_{\hat{x}}$ updates a subset of the values updated by ρ (this trivially follows from definition of $\rho_{\hat{x}}$).

□

Lemma 6.3 Let \mathcal{D} be a database scheme, \mathcal{AC} a set of aggregate constraints on \mathcal{D} , D an instance of \mathcal{D} such that $D \not\models \mathcal{AC}$, and $Q = P(a_1, \dots, a_n)$ a ground boolean query over \mathcal{D} . If there is a repair ρ for D w.r.t. \mathcal{AC} such that $Q \notin \rho(D)$, then there is a repair ρ' for D such that (i) $\lambda(\rho') \subseteq \lambda(\rho)$, (ii) $Q \notin \rho'(D)$ and (iii) ρ' has polynomial size w.r.t. D and Q .

Proof. If we assume that there is no tuple τ in P such that for each $A \in \mathcal{K}_P$, $\tau[A] = Q[A]$, then Q is false in every repair (recall that no repair can change key attributes in a relation). Moreover, Lemma 6.1 ensure that there is a repair ρ' such that $\lambda(\rho') \subseteq \lambda(\rho)$ and ρ' has polynomial size w.r.t. D . Therefore, since the size of ρ' does not depend of the size of Q , then it has also polynomial size w.r.t. D and Q .

Now assume that there is a tuple τ in P which coincides with Q in all the key attributes of P . Consider the system of inequalities $In(\mathcal{AC})$ obtained from \mathcal{AC} as explained in the proof of Lemma 6.1, where each variable $x_{t,A}$ corresponds to a pair $\langle t, A \rangle \in \lambda(\rho)$. Since $\tau \notin \rho(D)$, then there is at least a pair $\langle \tau, A \rangle \in \lambda(\rho)$, where $A \in \mathcal{M}_P$. We will denote as V^+ (resp. V^-) the set of variables x_{τ, A_i} such that there is an atomic update $\langle \tau, A_i, v_i \rangle \in \rho$ where $v_i > a_i$ (resp. $v_i < a_i$) holds. Let $In(\mathcal{AC})^+$ be the system of inequalities obtained adding to $In(\mathcal{AC})$ the inequalities:

- $x_{\tau, A_i} > a_i$, for each $x_{\tau, A_i} \in V^+$;
- $x_{\tau, A_i} < a_i$, for each $x_{\tau, A_i} \in V^-$.

Reasoning as in the prof of Claim 6.1 we can prove that $In(\mathcal{AC})^+$ has at least one solution \hat{x} of polynomial size w.r.t. D and Q . Moreover, reasoning as in the prof of Claim 6.2 it is easy to see that, let $\hat{x}_{t,A}$ be the value of variable $x_{t,A}$ in the solution \hat{x} of $In(\mathcal{AC})^+$, then the set of updates

$$\rho_{\hat{x}} = \{ \langle t, A, \hat{x}_{t,A} \rangle \mid \langle t, A \rangle \in \lambda(\rho) \wedge \hat{x}_{t,A} \neq t[A] \}$$

is a repair for D w.r.t. \mathcal{AC} such that $Q \notin \rho_{\hat{x}}(D)$ (the fact that $Q \notin \rho_{\hat{x}}(D)$ is guaranteed by the set of equations added to $In(\mathcal{AC})$). It is easy to see that $\rho_{\hat{x}}$ is of polynomial size w.r.t. D and Q too, and that $\rho_{\hat{x}}$ updates a subset of the values updated by ρ .

□

Theorem 6.3 (Consistent query answer under set-minimal semantics)

Let \mathcal{D} be a database scheme, D be an instance of \mathcal{D} , \mathcal{AC} a set of aggregate constraints on \mathcal{D} and Q a ground query over D . Deciding whether $Q^{set}(D) = true$ is Π_2^P -complete (w.r.t. the size of D).

Proof. Membership. Membership in Π_2^P can be proved showing that the complement of the consistent query answer problem under *set-minimal* semantics is in Σ_2^P . A polynomial size witness for the problem of deciding whether the consistent query answer is *false* is a *minimal* repair ρ such that $Q \notin \rho(D)$. If such a repair ρ exists, then Lemma 6.3 guarantees that a polynomial size repair ρ' such that $Q \notin \rho'(D)$ and $\lambda(\rho') \subseteq \lambda(\rho)$ exists too. Moreover, as shown in Theorem 6.2, testing whether a repair is *minimal* under set-minimal semantics can be accomplished by means of an *NP* oracle. Thus, the problem of deciding whether there is a minimal repair such the consistent query answer is *false* solved in polynomial time by nondeterministic Turing machines with an *NP* oracle.

Hardness. Hardness can be proved by showing a reduction from the following implication problem in the context of propositional logic over a finite domain V , which was shown to be Π_2^P -complete in [36]: “given an atomic knowledge base $T = \{a_1, \dots, a_n\}$, where a_1, \dots, a_n are atoms of V , an atom $q \in T$ and a formula p on V , decide whether q is derivable from every model in $T \circ_S p$ ”, where $T \circ_S p$ is the updated (or revised) knowledge base according to the Satoh’s revision operator.

Informally, Satoh’s revision operator \circ_S selects the models of p that are “closest” to models of T : closest models are those whose symmetric difference with models of T is minimal under set-inclusion semantics. In order to formally define the semantics of \circ_S we first introduce some preliminaries. Let $Mod(p)$ be the set of models of a formula p . Let $\Delta^{min}(T, p) = \min_{\subseteq}(\{M \Delta M' : M \in Mod(p), M' \in Mod(T)\})$, that is the family of \subseteq -minimal sets obtained as symmetric difference between models of p and T . The semantics of Satoh’s operator (i.e. the set of models of the knowledge base T revised according to the formula p) is defined as follows:

$$Mod(T \circ_S p) = \{ M \in Mod(p) : \exists M' \in Mod(T) \text{ s.t. } M \Delta M' \in \Delta^{min}(T, p) \}.$$

In the following the set of atoms occurring in p will be denoted as $V(p)$. Π_2^P -completeness of the implication problem was shown to hold also if $V(p) \subseteq T$ [36]: we consider this case in our proof. Observe that the definition of \circ_S entails that for each $M \in \Delta^{min}(T, p)$ it holds that $M \subseteq T \cap V(p)$, thus M is a subset of T .

We now consider an instance $\langle T, p, q \rangle$ of implication problem, where T is the atomic knowledge base $\{a_1, \dots, a_n\}$, p is a propositional formula (with $V(p) \subseteq T$), and q is an atom in T .

Let C_p be a boolean circuit equivalent to p . We consider the database scheme \mathcal{D} introduced in the hardness proof of Theorem 6.1. Moreover, we consider an instance D which is the translation of C_p obtained in the same way as Theorem 6.1, except that:

- relation *input* must contain not only the tuples corresponding to the inputs of C_p (i.e. the atoms in $V(p)$), but also the tuples corresponding to the atoms of $T \setminus V(p)$;

- for each tuple inserted in relation *input*, attribute *Val* is set to 1, which means assigning *true* to all the atoms of *T*.

Recall that measure attributes in the tuples of relations *gate* and *gateInput* are set to -1 (corresponding to an undefined truth value).

Let \mathcal{AC} be the same set of constraints used in the proof of Theorem 6.1. As explained in the hardness proof of Theorem 6.1, \mathcal{AC} defines the semantics of C_p and requires that C_p is true. Note that every repair ρ for D w.r.t. \mathcal{AC} must update all measure attributes that initially are set to -1 in D . Therefore, given two repairs ρ and ρ' , they differ only on the set of atomic updates performed on relation *input*.

Obviously, every *set*-minimal repair of ρ for D w.r.t. \mathcal{AC} corresponds to a model M in $Mod(T \circ_S p)$, and vice versa. In fact, given a *set*-minimal repair ρ for D w.r.t. \mathcal{AC} , a model M for $T \circ_S p$ can be obtained from the repaired database considering only the tuples in relation *input* where attribute *Val* is equal to 1 after applying ρ . Observe that the set of atoms M corresponding to ρ is a model $T \circ_S p$, otherwise there would exist $M' \subset M$ with $M' \in Mod(T \circ_S p)$, and the repair ρ' corresponding to M' would satisfy $\rho' <_{set} \rho$, thus contradicting the minimality of ρ . Likewise, it is easy to see that any model in $Mod(T \circ_S p)$ corresponds to a minimal repair for D w.r.t. \mathcal{AC} .

Finally consider the query $Q = input(id(q), 1)$. The above considerations suffice to prove that q is derivable from every model in $Mod(T \circ_S p)$ iff $input(id(q), 1)$ is true in $\rho(D)$ for every *set*-minimal repair ρ for D w.r.t. \mathcal{AC} , that is the consistent answer of $input(id(q), 1)$ on D w.r.t. \mathcal{AC} is true. \square

Theorem 6.4 (Consistent query answer under *card*-minimal semantics)

Let \mathcal{D} be a database scheme, D an instance of \mathcal{D} , \mathcal{AC} a set of aggregate constraints on \mathcal{D} and Q be a query over D . Deciding whether $Q^{card}(D) = true$ is $\Delta_2^P[\log n]$ -complete (w.r.t. the size of D).

Proof. Membership. Membership in $\Delta_2^P[\log n]$ derives from the fact that repairs on D can be partitioned into the two sets T and F consisting of all repairs ρ_i s.t. $Q \in \rho_i(D)$ and, respectively, $Q \notin \rho_i(D)$. Let $MinSize(T) = \min_{\rho \in T}(|\lambda(\rho)|)$, and $MinSize(F) = \min_{\rho \in F}(|\lambda(\rho)|)$. It can be shown that $Q^{card}(D) = true$ iff $MinSize(T) < MinSize(F)$. Lemma 6.2 (resp. Lemma 6.3) ensures that an *NP*-oracle can be used for deciding whether there exists a repair ρ for D such that $Q \in \rho_i(D)$ (resp. $Q \notin \rho_i(D)$) and $|\lambda(\rho)|$ is equal to a given value between one and n . Thus, both $MinSize(T)$ and $MinSize(F)$ can be evaluated by a logarithmic number of *NP*-oracle invocations.

Hardness. Hardness can be proved by showing a reduction from the following implication problem in the context of propositional logic over a finite domain V : “given an atomic knowledge base T on V , a formula q on T and a formula p on V , decide whether q is derivable from every model in $T \circ_D p$ ”, where $T \circ_D p$ is the updated (or revised) knowledge base according to the Dalal’s revision operator. $\Delta_2^P[\log n]$ -completeness of this problem was shown in [36].

The semantics of Dalal’s revision operator is as follows. The models of $T \circ_D p$ are the models of p whose symmetric difference with models of T has minimum cardinality w.r.t. all other models of p . More formally, let $|\Delta^{min}(T, p)| = \min\{|M \Delta M'| : M \in Mod(p), M' \in Mod(T)\}$, that is the minimum number of atoms in which models of T and p diverge. Then models of $T \circ_D p$ are given by:

$$Mod(T \circ_D p) = \{M \in Mod(p) : \exists M' \in Mod(T) \text{ s.t. } |M \Delta M'| \in |\Delta^{min}(T, p)|\}.$$

Consider an instance $\langle V, T, p, q \rangle$ of the implication problem, where V is the finite domain of atoms, T an atomic knowledge base on V , p a formula on V , and q a formula on T . Let $V(p)$ and $V(q)$ denote the set of atoms of V occurring in p and q , respectively. Sets T , $V(p)$ and $V(q)$ can be partitioned into A, B, C, D, E , as shown in Fig. 6.1(a).

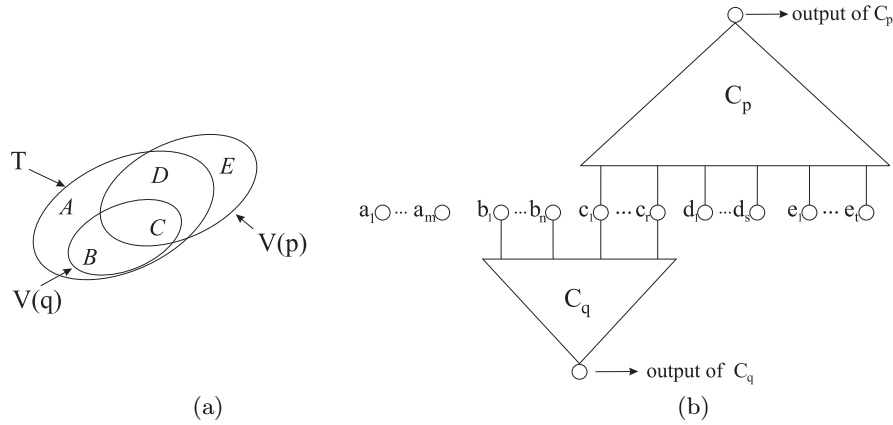


Fig. 6.1. (a) The partitioning of $T, V(p), V(q)$; (b) Circuits

Let C_p and C_q be two boolean circuits equivalent to p and q , respectively. C_p and C_q are reported in Fig. 6.1(b), with their inputs. In this figure, atoms belonging to $T, V(p)$ and $V(q)$ are represented as circles, and the two circuits are represented by means of triangles. In particular, inputs of C_q are the atoms b_1, \dots, b_n of B and the atoms c_1, \dots, c_r of C , whereas inputs of C_p are the atoms c_1, \dots, c_r of C , the atoms d_1, \dots, d_s of D , and the atoms e_1, \dots, e_t of E . That is, the atoms of C are inputs of both C_p and C_q .

These circuits can be represented as an instance of the database scheme \mathcal{D} introduced in the hardness proof of Theorem 6.1. In particular, we consider an instance D of \mathcal{D} which is the translation of C_p and C_q obtained in the same way as Theorem 6.1, except that:

- relation *input* contains a tuple for each atom in $A \cup B \cup C \cup D \cup E$;
- for each tuple inserted in relation *input*, attribute *Val* is set to 1 if it refers to an atom in T , -1 otherwise. This means assigning *true* to all the atoms of T , and an undefined truth value to atoms in E .

Recall that measure attributes in the tuples of relations $gate$ and $gateInput$ are set to -1 .

We consider the set of aggregate constraints \mathcal{AC} consisting of constraints 1-5 introduced in the hardness proof of Theorem 6.1, plus the aggregate constraint $NORVal(id(o_p)) = 1$, where $id(o_p)$ is the identifier of the output gate of C_p . As explained in the hardness proof of Theorem 6.1, \mathcal{AC} defines the semantics of C_p and C_q and requires that C_p is true.

Note that every repair ρ for D w.r.t. \mathcal{AC} must update all value attributes that initially are assigned -1 in D . Therefore, given two repairs ρ and ρ' for D w.r.t. \mathcal{AC} , they differ only on the number of atomic updates performed on the tuples of $input$ where Val was set to 1 in D .

Obviously, every *card*-minimal repair of ρ for D w.r.t. \mathcal{AC} corresponds to a model M in $Mod(T \circ_D p)$, and vice versa (this can be proven straightforwardly, analogously to the proof of Theorem 6.3, where the correspondence between *set*-minimal repairs for D and models of $T \circ_S p$ has been shown).

Finally consider the query $Q = input(id(o_q), 1)$, where o_q denotes the the output gate of C_q . The above-mentioned considerations suffice to prove that q is derivable from every model in $Mod(T \circ_D p)$ iff $input(id(o_q), 1)$ is true in $\rho(D)$ for every *card*-minimal repair ρ for D w.r.t. \mathcal{AC} , that is the consistent answer of $input(id(o_q), 1)$ on D w.r.t. \mathcal{AC} is true. □

6.6 Discussion

In this chapter we have addressed the problem of repairing and extracting reliable information from numerical databases violating *aggregate constraints*. We fill a gap in previous works dealing with inconsistent data, where only traditional forms of constraints were considered (as shown in the previous chapters). In fact, aggregate constraints frequently occur in many real-life scenarios where guaranteeing the consistency of numerical data is mandatory. In particular, we have considered aggregate constraints defined as sets of linear inequalities on aggregate-sum queries on input data. For this class of constraints we have characterized the complexity of several issues related to the computation of consistent query answers.

All the approaches discussed in Chapter 3 and Chapter 4 assume that tuple insertions and deletions are the basic primitives for repairing inconsistent data. On the other hand, approaches examined in Chapter 5 provide repairs consisting also of value-update operations. Specifically, the repairing strategy presented in this chapter is similar to those introduced in [42] and [16] (cfr. Section 5.1 and Section 5.2, respectively). These approaches define repairs working at the attribute-value level and such that the values for the attributes in the key of the relations hold steady. But, these techniques are not well-suited in the contexts like those of Example 6.1, as they do not provide solutions for

repairing and querying data which are inconsistent with respect to a set of aggregate constraints.

The first work investigating aggregate constraints on numerical data is [71], where the consistency problem of very general forms of aggregation is considered, but no issue related to data-repairing is investigated. In [16] the problem of repairing databases by fixing numerical data at attribute level is investigated. The authors shown that deciding the existence of a repair under both denial constraints (where built-in comparison predicates are allowed) and a *non-linear* form of multi-attribute aggregate constraints is undecidable (cfr. Section 5.2.2). Then they disregarded aggregate constraints and focused on the problem of repairing data violating denial constraints, where no form of aggregation is allowed in the adopted constraints. In this chapter we shown that the repair existence problem is decidable in presence of aggregate constraints (cfr. Theorem 6.1). Basically, decidability results from the linear nature of the considered constraints, where products between two *measure attributes* in attribute expressions of the aggregation functions were not allowed.

Computing Repairs for Inconsistent Numerical Data

In the previous chapter, the problem of extracting consistent information from relational databases violating integrity constraints on numerical data has been addressed. Specifically, aggregate constraints defined as linear inequalities on aggregate-sum queries on input data have been considered. In this chapter we provide an architecture providing robust data acquisition facilities from input documents containing tabular data. This architecture is based on the data-repairing framework presented in Chapter 6. We exploit integrity constraints defined on the input data to support the detection and the repair of inconsistencies in the data arising from errors occurring in the acquisition phase. Specifically, we will introduce a specific but expressive form of *aggregate* constraints, namely *steady aggregate constraints*, which enables the computation of a repair to be expressed as a Mixed-Integer Linear Programming problem.

7.1 Introduction

The need to acquire data from different sources of information often arises in many application scenarios, such as e-procurement, competitor analysis, business intelligence. In several cases these sources are heterogenous documents, possibly represented according to different formats, ranging from paper documents to electronic ones (PDF, MSWord, HTML files). In order to be exploited to provide valuable knowledge, information must be extracted from the original documents and re-organized into a machine-readable format. The problem of defining efficient and effective approaches accomplishing this task is a challenging issue in the context of Information Extraction (IE) [59]. Most of traditional IE techniques focus on efficiency, providing unsupervised extraction algorithms which automatically extract records from documents. However, it frequently happens that some of the extracted records are not correctly recognized, i.e. the value of one (or more) field has been misspelled. In several contexts (such as balance analysis) extracted information must be

100% error free in order to be profitably exploited, thus unsupervised approaches are not well-suited. In these cases, data transcription from input documents into a machine-readable format requires massive human intervention, thus compromising efficiency and making valuable resources be wasted. Human intervention is mainly devoted to verifying the correctness of acquired data by comparing them with the content of source documents.

Indeed, if integrity constraints are defined on the input data, this kind of human intervention can be reduced by automatically verifying whether acquired data satisfy these constraints, thus limiting manual corrections to those pieces of acquired data which do not satisfy them. In fact current approaches exploiting integrity constraints on source documents require inconsistent acquired data to be manually edited by a human operator. This editing task is likely to be onerous, since a large amount of data in the input documents need to be accessed and compared with the acquired ones.

We observe that human intervention can be reduced by exploiting some repairing technique to suggest the “most likely” way of fixing inconsistent data. We introduce the architecture of a system (namely, *DART* - Data Acquisition and Repairing Tool) based on this observation. The contribution provided by this system can be better understood after reading the following example, describing a specific application scenario (that is, data acquisition from balance sheets).

Example 7.1 The balance sheet is a financial statement of a company providing information on what the company owns (its assets), what it owes (its liabilities), and the value of the business to its stockholders. A thorough analysis of a company balance sheet is extremely important for both stock and bond investors, since it allows potential liquidity problems to be detected, thus determining the company financial reliability as well as its ability to satisfy financial obligations.

Fig. 7.1 is a portion of a document containing two *cash budgets* for a firm, each of them related to a year. Each cash budget is a summary of cash flows (receipts, disbursements, and cash balances) over the specified periods.

This cash budget satisfies the following integrity constraints:

- a) for each year, the sum of *cash sales* and *receivables* in section *Receipts* must be equal to *total cash receipts*;
- b) for each year, the sum of *payment of accounts*, *capital expenditure* and *long-term financing* must be equal to *total disbursements* (in section *Disbursements*);
- c) for each year, the *net cash inflow* must be equal to the difference between *total cash receipts* and *total disbursements*;
- d) for each year, the *ending cash balance* must be equal to the sum of the *beginning cash* and the *net cash inflow*;

Generally balance sheets like the ones depicted in Fig. 7.1 are available as paper documents, thus they cannot be automatically processed by balance

2003	Receipts	beginning cash	20
		cash sales	100
		receivables	120
		total cash receipts	220
	Disbursements	payment of accounts	120
		capital expenditure	0
		long-term financing	40
		total disbursements	160
Balance	net cash inflow	60	
	ending cash balance	80	
2004	Receipts	beginning cash	80
		cash sales	100
		receivables	100
		total cash receipts	200
	Disbursements	payment of accounts	130
		capital expenditure	40
		long-term financing	20
		total disbursements	190
Balance	net cash inflow	10	
	ending cash balance	90	

Fig. 7.1. An input document

analysis tools, since these work only on electronic data. In fact, some companies do business acquiring electronic balance data and reselling them in a format suitable for being processed by commercial analysis tools. Currently electronic versions are obtained by means of either human transcriptions or OCR acquisition tools. Both these approaches are likely to result in erroneous acquisition, thus compromising the reliability of the analysis task.

An example of numerical value recognition error occurring during the acquisition phase is the recognition of the value 250 instead of 220 for “total cash receipts” in the year 2003. Consequently, some constraints are not satisfied on the acquired data for year 2003:

- i) in section *Receipts*, the value of *total cash receipts* is not equal to the sum of values of *cash sales* and *receivables*;
- ii) the value of *net cash inflow* is not to equal the difference between *total cash receipts* and *total disbursements*.

Furthermore, some symbol recognition errors in non-numerical strings may occur in the acquisition phase. For instance, the item “bgnning cesh” could be recognized instead of “beginning cash”.

□

DART is a system supporting the acquisition of heterogeneous documents and the supervised repairing of the acquired data. With respect to Example 7.1, DART will suggest to change the “total cash receipts” value for year

2003 from 250 (i.e. the acquired value) to 220, thus reducing the human intervention, as the human operator is no longer required to access the whole input document to fix acquisition errors making integrity constraints violated. In particular, DART is based on the notion of *card*-minimal repair introduced in Section 6.4.2 for repairing numerical data which are inconsistent w.r.t. aggregate constraints. Aggregate constraints can express constraints like those defined in the context of balance-sheet data. The notion of *card*-minimal repair is well-suited for these contexts, where data inconsistency is due to bad symbol recognition during the acquisition phase. Indeed, applying the *card*-minimal semantics means searching for repairs changing the minimum number of acquired values, which corresponds to the assumption that the minimum number of errors occurred in the acquisition phase.

In the scenario of balance-sheet acquisition, the relevant information is formatted according to a tabular layout. Therefore, we introduce a system architecture aiming at supervised acquiring of information encoded into tabular data. However, observe that this feature does not limit DART to the acquisition of balance sheets, as tabular data often occur in many different application contexts, such as web sites publishing product catalogs.

The system described in this chapter embeds a wrapping module for extracting information from tabular data. This module can manage tables having “variable” structures, i.e. tables whose cells can span multiple rows and columns, according to no pre-determined scheme. Moreover, a framework for computing *card*-minimal repairs on wrongly acquired data is introduced to drive the data validation process. This framework exploits a specific form of aggregate constraints, namely, *steady aggregate constraints*, defined on the source documents to check the consistency of the acquired data and computing a repair.

Describing the wrapping technique in detail is out of the scope of this chapter. Here we will focus on presenting the architecture of the system and the technique adopted for computing repairs.

7.2 DART a Data Acquisition and Repairing Tool

DART (*Data Acquisition and Repairing Tool*) is a system providing robust data acquisition facilities. It takes as input documents containing tabular data, and it exploits integrity constraints defined on the input data to support the detecting and the repairing of inconsistencies due to errors occurring in the acquisition phase. If acquisition errors are detected, the system proposes a way to correct these errors. Proposed corrections are validated by means of human intervention. In order to detect and repair inconsistencies, integrity constraints are considered expressing algebraic relations among the numerical data reported in the cells of the input tables. These constraints are exploited only to fix the acquired numerical values. Moreover, a dictionary of the terms

used in the specific scenario which the input documents refer to is exploited to provide spelling error corrections on non-numerical strings.

Two kinds of user interact with DART, namely the *acquisition designer* and the *operator*. The former is an expert on the application context and specifies the metadata which are used to support both the extraction of tabular data and the repairing process. The latter interacts with the system during the acquisition of each document: if the acquired data need to be corrected, he is prompted to validate proposed corrections.

As shown in Fig. 7.2, DART consists of two macro-modules. The first module takes as input documents containing tabular data and returns a relational database where the extracted tabular data are stored. It performs three steps: it loads the input document and convert it in HTML format, it extracts the tabular data from the HTML document and it transforms them into a database instance. This module exploits metadata specified by the acquisition designer, which describe the structure and the semantics of the input documents¹.

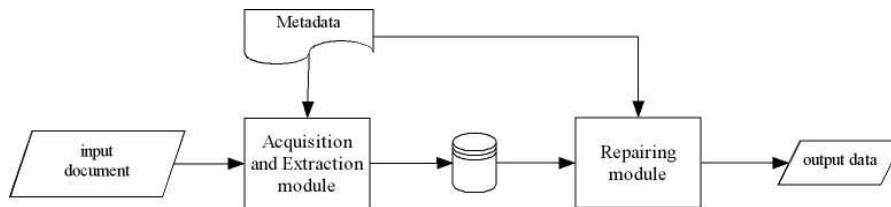


Fig. 7.2. Data flow in DART

The second module takes as input the database instance D generated by the *acquisition and extraction module*. It locates possible inconsistencies in D and returns a repair for D . Both the inconsistency detection and the repair computation are accomplished according to a set of aggregate constraints \mathcal{AC} defined by acquisition designer and represented in the metadata. In more detail, the *repairing module* transforms the problem of finding a *card*-minimal repair for D w.r.t. \mathcal{AC} into an MILP instance (Mixed-Integer Linear Programming problem) and solves it providing a repair for D . The proposed repair is then validated by the operator, who either accepts it or requires to compute a different repair. In fact, it can be the case that the proposed repair is unsatisfactory since the operator realizes that it consists of value updates which do not correspond to the actual content of the source document. In this case the operator inserts further constraints on the acquired data. Basically, he drives

¹ As it will be clear in the following, designing an extraction module taking as input HTML documents will make it possible to exploit its features also in Web applications, where the problem of automatically extracting information from HTML pages often arises in many scenarios.

the repairing process by specifying the exact values that some pieces of the repaired data must take.

Example 7.2 Consider the database scheme \mathcal{D} consisting of the single relation scheme $CashBudget(Year, Section, Subsection, Type, Value)$, and its instance reported in Fig. 7.3. This instance represents a possible output of the *acquisition and extraction module* when DART takes as input the document in Fig. 7.1 (it results from the case that a symbol recognition error occurred in the acquisition phase, so that the acquired value of *total cash receipts* is 250 instead of 220). Values ‘*det*’, ‘*aggr*’ and ‘*drv*’ in column *Type* stand for *detail*, *aggregate* and *derived*, respectively. In particular, an item of the table is *aggregate* if it is obtained by aggregating items of type *detail* of the same section, whereas a *derived* item is an item whose value can be computed using the values of other items of any type and belonging to any section.

<i>Year</i>	<i>Section</i>	<i>Subsection</i>	<i>Type</i>	<i>Value</i>
2003	Receipts	beginning cash	drv	20
2003	Receipts	cash sales	det	100
2003	Receipts	receivables	det	120
2003	Receipts	total cash receipts	aggr	250
2003	Disbursements	payment of accounts	det	120
2003	Disbursements	capital expenditure	det	0
2003	Disbursements	long-term financing	det	40
2003	Disbursements	total disbursements	aggr	160
2003	Balance	net cash inflow	drv	60
2003	Balance	ending cash balance	drv	80
2004	Receipts	beginning cash	drv	80
2004	Receipts	cash sales	det	100
2004	Receipts	receivables	det	100
2004	Receipts	total cash receipts	aggr	200
2004	Disbursements	payment of accounts	det	130
2004	Disbursements	capital expenditure	det	40
2004	Disbursements	long-term financing	det	20
2004	Disbursements	total disbursements	aggr	190
2004	Balance	net cash inflow	drv	10
2004	Balance	ending cash balance	drv	90

Fig. 7.3. A cash budget

Constraints a) and b) defined in Example 7.1 can be expressed as: for each section and year, the sum of the values of all *detail* items must be equal to the value of the *aggregate* item of the same section and year. Therefore, they can be expressed by the following aggregate constraint (cfr. Definition 6.1):

Constraint 1:

$$\text{CashBudget}(y, x, -, -, -) \implies \chi_1(x, y, 'det') - \chi_1(x, y, 'aggr') = 0$$

where χ_1 and χ_2 are the following aggregation functions are defined on the relation scheme *CashBudget*:

$$\begin{array}{ll} \chi_1(x, y, z) = \text{SELECT } \text{sum(Value)} & \chi_2(x, y) = \text{SELECT } \text{sum(Value)} \\ \text{FROM } \text{CashBudget} & \text{FROM } \text{CashBudget} \\ \text{WHERE } \text{Section} = x & \text{WHERE } \text{Year} = x \\ \text{AND Year} = y \text{ AND Type} = z & \text{AND Subsection} = y \end{array}$$

Function χ_1 returns the sum of *Value* of all the tuples having *Section* x , *Year* y and *Type* z . Function χ_2 returns the sum of *Value* of all the tuples where *Year*= x and *Subsection*= y . In our example, as the pair *Year, Subsection* is a key for the tuples of *CashBudget*, the sum returned by χ_2 is an attribute value of a single tuple.

Constraints c) and d) of Example 7.1 can be expressed as follows:

Constraint 2:

$$\begin{array}{l} \text{CashBudget}(x, -, -, -, -) \implies \chi_2(x, 'net\ cash\ inflow') - \\ (\chi_2(x, 'total\ cash\ receipts') - \chi_2(x, 'total\ disbursements')) = 0 \end{array}$$

Constraint 3:

$$\begin{array}{l} \text{CashBudget}(x, -, -, -, -) \implies \chi_2(x, 'ending\ cash\ balance') - \\ (\chi_2(x, 'beginning\ cash') + \chi_2(x, 'net\ cash\ inflow')) = 0 \end{array}$$

As seen in Section 6.4.2, the unique *card*-minimal repair ρ for *CashBudget* w.r.t. constraints 1), 2) and 3) consists in decreasing attribute *Value* in the tuple: $t = \text{CashBudget}(2003, 'Receipts', 'total\ cashreceipts', 'aggr', 250)$ down to 220; that is, $\rho = \{ < t, Value, 220 > \}$. □

7.3 Steady Aggregate Constraints

In this section we introduce a restricted form of aggregate constraints, namely *steady aggregate constraints*. On the one hand, steady aggregate constraints are less expressive than (general) aggregate constraints, but, on the other hand, computing a *card*-minimal repair w.r.t. a set of steady aggregate constraints can be accomplished by solving an instance of an MILP (Mixed Integer Linear Programming) problem. This allows us to adopt standard techniques addressing MILP problems to accomplish the computation of a *card*-minimal repair (as it will be clear in the following, this would not be possible for general aggregate constraints). However, observe that the loss in expressiveness is not dramatic, as steady aggregate constraints suffice to express relevant integrity constraints in many real-life scenarios. For instance, all the aggregate

constraints introduced in Example 7.2 can be expressed by means of steady aggregate constraints.

Before providing the formal definition of steady aggregate constraint, we introduce some preliminary notations.

Given a relation scheme $P(A_1, \dots, A_n)$ and a conjunction of atoms ϕ containing the atom $P(x_1, \dots, x_n)$, we say that the attribute A_j *corresponds* to the variable x_j , for each $j \in [1..n]$. Given an aggregation function χ_i , we will denote as $\mathcal{W}(\chi_i)$ the union of the set of the attributes appearing in the WHERE clause of χ_i and the set of attributes corresponding to variables appearing in the WHERE clause of χ_i . Given an aggregate constraint κ where the aggregation functions χ_1, \dots, χ_n occur, we will denote as $\mathcal{A}(\kappa)$ the set of attributes $\bigcup_{i=1}^n \mathcal{W}(\chi_i)$. Given an aggregate constraint κ , we will denote as $\mathcal{J}(\kappa)$ the set of attributes such that for each $A \in \mathcal{J}(\kappa)$ there are two atoms $P_i(x_{i_1}, \dots, x_{i_n})$ and $P_j(x_{j_1}, \dots, x_{j_m})$ in $\phi(x_1, \dots, x_k)$ satisfying both the following conditions:

1. there are $i_l \in [i_1..i_n]$ and $j_h \in [j_1..j_m]$ such that $x_{i_l} = x_{j_h}$;
2. A corresponds to either x_{i_l} or x_{j_h} .

Basically, $\mathcal{J}(\kappa)$ contains attributes A corresponding to variables shared by two atoms in ϕ .

The reason why sets $\mathcal{A}(\kappa)$ and $\mathcal{J}(\kappa)$ have been introduced is that they allow us to detect a useful property. In fact, in the case that $\mathcal{A}(\kappa) \cup \mathcal{J}(\kappa)$ does not contain any *measure attribute* (i.e. attributes representing measure values, cfr. Section 6.2), the tuples in the database instance D which are “involved” in κ (i.e. the tuples where ϕ and the WHERE clauses of the aggregation functions in κ evaluate to true) can be detected without looking at the values of their measure attributes. As it will be clear in the following, if this syntactic property holds we can translate κ into a set of linear inequalities and then express the computation of a *card*-minimal repair w.r.t. κ as an instance of MILP.

Definition 7.1 (Steady aggregate constraint) Let \mathcal{D} be a database scheme, $\mathcal{M}_{\mathcal{D}}$ the set of measure attributes of \mathcal{D} and κ an aggregate constraint on \mathcal{D} . The aggregate constraint κ is said to be a *steady aggregate constraint* if:

$$(\mathcal{A}(\kappa) \cup \mathcal{J}(\kappa)) \cap \mathcal{M}_{\mathcal{D}} = \emptyset$$

□

Example 7.3 Consider a database scheme \mathcal{D} containing the relation schemes $P_1(A_1, A_2, A_3)$ and $P_2(A_4, A_5, A_6)$, where $\mathcal{M}_{\mathcal{D}} = \{A_2, A_4\}$. Let κ be the following aggregate constraint on \mathcal{D} :

$$\forall x_1, x_2, x_3, x_4, x_5 [P_1(x_1, x_2, x_3), P_2(x_3, x_4, x_5) \implies \chi(x_2) \leq K]$$

where:

$$\chi(x) = \text{SELECT sum}(A_6) \\ \text{FROM } P_2 \\ \text{WHERE } A_5 = x$$

We have that $\mathcal{A}(\kappa) = \{A_5, A_2\}$ and $\mathcal{J}(\kappa) = \{A_3, A_4\}$, therefore κ is not a steady aggregate constraint.

Consider *Constraint 1* of Example 7.2. We have that $\mathcal{A}(\text{Constraint 1}) = \{\text{Year}, \text{Section}, \text{Type}\}$ and $\mathcal{J}(\text{Constraint 1}) = \emptyset$. Since $\mathcal{M}_{\mathcal{D}} = \{\text{Value}\}$, Constraint 1 is a steady aggregate constraint. Similarly, it is straightforward to show that also constraints 2) and 3) are steady aggregate constraints. \square

7.3.1 Complexity Results under Steady Aggregate Constraints

All complexity results characterizing either the repair existence problem and the consistent query answer problem given in Chapter 6 (where general aggregate constraints were considered) are still valid for the class of steady aggregate constraints.

Theorem 7.1 (Repair existence) Let \mathcal{D} be a database scheme, \mathcal{AC} a set of steady aggregate constraints on \mathcal{D} , and D an instance of \mathcal{D} such that $D \not\models \mathcal{AC}$. The problem of deciding whether there is a repair for D is NP-complete (w.r.t. the size of D).

Proof. Membership. It is straightforward that membership proof of Theorem 6.1 still holds for steady aggregate constraints.

Hardness. As in hardness proof of Theorem 6.1, we show a reduction from CIRCUIT SAT to our problem. But in this case we have to use only steady aggregate constraints. Thus, the hardness proof of Theorem 6.1 need to be revisited since also non-steady aggregate constraints was used for defining the semantics of the boolean circuit.

We consider the database scheme \mathcal{D} introduced in the hardness proof of Theorem 6.1. Moreover, we consider an instance D which is the translation of a boolean circuit C (consisting of only NOR gates) obtained in the same way as Theorem 6.1.

Let \mathcal{AC}^* be the set of aggregate constraints used in the proof of Theorem 6.1. All the constraints in \mathcal{AC}^* are steady aggregate constraints except that the constraint: $\text{ValidInput}(\) + \text{ValidGate}(\) = 0$. It entails that only 0 and 1 can be assigned either to attributes *orVal* and *norVal* in relation *gate*, and to attribute *Val* in relation *input*.

This constraint can be expressed by a fixed set of steady aggregate constraints as follows. Assume that the domain of the attributes *norVal*, *orVal* and *Val* in relation *input* is the infinite domain of integers \mathbb{Z} . The following set of steady aggregate constraints on \mathcal{D} are equivalent to the aggregate $\text{ValidInput}(\) + \text{ValidGate}(\) = 0$.

- 1) $input(X, -) \implies InputVal(X) \leq 1$
- 2) $input(X, -) \implies InputVal(X) \geq 0$
- 3) $gate(X, -, -) \implies NORVal(X) \leq 1$
- 4) $gate(X, -, -) \implies NORVal(X) \geq 0$
- 5) $gate(X, -, -) \implies ORVal(X) \leq 1$
- 6) $gate(X, -, -) \implies ORVal(X) \geq 0$

Specifically, as the domain of attribute Val is \mathbb{Z} , constraint a) and b) entail that, for each tuple in relation $input$, the value of Val must be in $\{0, 1\}$. Similarly, constraint c) and d) (resp. e) and f)) imply that, for each tuple in the relation $gate$, the value of $norVal$ (resp. $orVal$) must be in $\{0, 1\}$.

Let \mathcal{AC} be the set of steady aggregate constraints consisting of the constraints in \mathcal{AC}^* except that the constraint $ValidInput(\) + ValidGate(\) = 0$ is replaced with the constraints 1) – 6) above.

As explained in the hardness proof of Theorem 6.1, \mathcal{AC} defines the semantics of the boolean circuit C and requires that C is true. Therefore, deciding whether there is a truth assignment which evaluates C to *true* is equivalent to asking whether there is a repair ρ for D w.r.t. \mathcal{AC} . □

In Section 6.5 the complexity characterization of the consistent query answer problem under *card*-minimal semantics has been provided. As stated in the following corollary, the result obtained for (general) aggregate constraints still holds for steady aggregate constraints.

Corollary 7.1 Let \mathcal{D} be a database scheme, D an instance of \mathcal{D} , \mathcal{AC} a set of steady aggregate constraints on \mathcal{D} and Q be a boolean ground query over D . Deciding whether Q evaluates to *true* in D is $\Delta_2^p[\log n]$ -complete (w.r.t. the size of D).

Proof. It is easy to see that the membership proof of Theorem 6.4 still holds for steady aggregate constraints.

The hardness proof of Theorem 6.4 is based on the construction of a database scheme \mathcal{D} representing a boolean circuit. A set of aggregate constraints expressing the semantics of the circuit is defined. As shown in Theorem 7.1, this construction can be accomplished using only steady aggregate constraints. Thus, we can use the same reduction of Theorem 6.4 except that, as in Theorem 7.1, we replace the (general) aggregate constraint $ValidInput(\) + ValidGate(\) = 0$ with the set of steady aggregate constraints 1) – 6) shown in the hardness proof of Theorem 7.1. □

7.4 Computing a *Card*-Minimal Repair

In several application scenarios, such as that described in Example 7.1, we are more interested in computing a repair (fixing all the acquired values) than

evaluating whether a single acquired value is “reliable”. In this section we define a technique for computing a *card*-minimal repair for a database w.r.t a set of steady aggregate constraints, which is based on the translation of the repair-evaluation problem into an instance of a mixed-integer linear programming (MILP) problem [48]. Our technique exploits the restrictions imposed on steady aggregate constraints w.r.t. general aggregate constraints to accomplish the computation of a repair. As it will be clear later, this approach does not work for (general) aggregate constraints.

Consider a database scheme \mathcal{D} and a set of steady aggregate constraints \mathcal{AC} on \mathcal{D} . In this case, we can model the problem of finding a *card*-minimal repair as MILP problem (if the domain of numerical attributes is restricted to \mathbb{Z} then it can be formulated as an ILP problem).

We first show how a steady aggregate constraint can be expressed by a set of linear inequalities.

Consider the steady aggregate constraint κ :

$$\forall x_1, \dots, x_k \left(\phi(x_1, \dots, x_k) \implies \sum_{i=1}^n c_i \cdot \chi_i(y_{i_1}, \dots, y_{i_{m_i}}) \leq K \right)$$

where $\bigcup_{i=1}^n \{y_{i_1}, \dots, y_{i_{m_i}}\}$ is a subset of $\{x_1, \dots, x_k\}$ and for each $i \in \{1, \dots, n\}$:

$$\begin{aligned} \chi_i(y_{i_1}, \dots, y_{i_{m_i}}) &= \text{SELECT sum}(e_i) \\ &\text{FROM } P_{\chi_i} \\ &\text{WHERE } \alpha_i(y_{i_1}, \dots, y_{i_{m_i}}) \end{aligned}$$

Without loss of generality, we assume that each attribute expression e_i occurring in the aggregation function χ_i is either an attribute or a constant.

We associate a variable z_{t, A_j} to each database value $t[A_j]$, where t is a tuple in the database instance D and A_j is an attribute in $\mathcal{M}_{\mathcal{D}}$, i.e. the set of measure attributes of D . The variable z_{t, A_j} is defined on the same domain as A_j . For every ground substitution θ of x_1, \dots, x_k such that $\phi(\theta x_1, \dots, \theta x_k)$ is *true*, we will denote as T_{χ_i} the set of the tuples involved in the aggregation function χ_i , that is $T_{\chi_i} = \{t : t \models \alpha_i(\theta y_{i_1}, \dots, \theta y_{i_{m_i}})\}$.

The translation of χ_i , denoted as $\mathcal{P}(\chi_i)$, is defined as follows:

$$\mathcal{P}(\chi_i) = \begin{cases} \sum_{t \in T_{\chi_i}} z_{t, A_j} & \text{if } e_i = A_j; \\ e_i \cdot |T_{\chi_i}| & \text{if } e_i \text{ is a constant.} \end{cases}$$

Starting from $\mathcal{P}(\chi_i)$, the whole constraint κ can be expressed as a set \mathcal{S} of linear inequalities as follows. For every ground substitution θ of x_1, \dots, x_k such that $\phi(\theta x_1, \dots, \theta x_k)$ is true, \mathcal{S} contains the following inequality:

$$\sum_{i=1}^n c_i \cdot \mathcal{P}(\chi_i) \leq K$$

Observe that this construction is not possible for a non-steady aggregate constraint since, given a database instance D and an aggregation function χ_i in the constraint, we cannot determine T_{χ_i} : changing a measure value might result in changing the set of the tuples involved the aggregation function.

For the sake of simplicity, in the following we associate to each pair $\langle t, A_j \rangle$ an integer index i , therefore we write z_i instead of z_{t, A_j} . If we assume that the number of values involved in constraints in \mathcal{AC} concerning the given database instance D is N then the index i will take values in $[1..N]$.

As shown above, we can translate each steady aggregate constraint into a system linear inequalities. The translation of all aggregate constraints in \mathcal{AC} produces the system of linear inequalities $A \cdot Z \leq B$, where $Z = [z_1, z_2, \dots, z_N]^T$. This system will be denoted as $\mathcal{S}(\mathcal{AC})$.

Example 7.4 Consider the database scheme \mathcal{D} consisting of the single relation scheme $CashBudget(Year, Section, Subsection, Type, Value)$, and its instance reported in Fig. 7.3. Assume that the set of aggregate constraints \mathcal{AC} consist of constraints 1), 2) and 3) of Example 7.1. The values involved in constraints in \mathcal{AC} w.r.t. the given database instance in Fig. 7.3 are as many as the number of tuples, that is $N = 20$. Therefore, z_i , ($1 \leq i \leq 20$) is the variable associated to the database value $t[Value]$, where t is the i -th tuple in Fig. 7.3. For instance, z_2 is the variable associated with the value of attribute *Value* in the tuple $t = CashBudget(2003, 'Receipts', 'cash sales', 'det', 100)$.

The translation of constraints 1), 2) and 3) is the following, where we explicitly write equalities instead of inequalities:

$$1) \begin{cases} z_2 + z_3 = z_4 \\ z_5 + z_6 + z_7 = z_8 \\ z_{12} + z_{13} = z_{14} \\ z_{15} + z_{16} + z_{17} = z_{18} \end{cases} \quad 2) \begin{cases} z_4 - z_8 = z_9 \\ z_{14} - z_{18} = z_{19} \end{cases} \quad 3) \begin{cases} z_1 - z_9 = z_{10} \\ z_{11} - z_{19} = z_{20} \end{cases}$$

$\mathcal{S}(\mathcal{AC})$ consists of the system obtained by assembling all the equalities reported above (basically, it is the intersection of the three systems above). \square

In the following we will denote the current database value corresponding to the variable z_i as v_i . That is, if z_i is associated with $t[A_j]$, then $v_i = t[A_j]$. Every solution s of $\mathcal{S}(\mathcal{AC})$ corresponds to a (possibly non-minimal) repair $\rho(s)$ of D w.r.t. \mathcal{AC} . Specifically, for each variable z_i which is assigned a value different from v_i , repair $\rho(s)$ contains an atomic update assigning the value z_i to the database item corresponding to z_i .

In order to decide whether a solution s of $\mathcal{S}(\mathcal{AC})$ corresponds to a *card*-minimal repair, we must count the number of variables of s which are assigned a value different from the corresponding source value in D . This is accomplished as follows. For each $i \in \{1, \dots, N\}$, we define a variable $y_i = z_i - v_i$ on the same domain as z_i . Consider the following system of linear inequalities, which will be denoted as $\mathcal{S}'(\mathcal{AC})$:

$$\begin{cases} AZ \leq B \\ y_i = z_i - v_i \quad \forall i \in \{1, \dots, N\} \end{cases}$$

As shown in [67], if a system of equalities has a solution, it has also a solution where each variable takes a value in $[-M, M]$, where M is a constant equal to $n \cdot (ma)^{2m+1}$, where m is the number of equalities, n is the number of variables and a is the maximum value among the modules of the system coefficients. It is straightforward to see that $\mathcal{S}'(\mathcal{AC})$ can be translated into a system of linear equalities in augmented form with $m = N+r$ and $n = 2 \cdot N+r$, where r is the number of rows of A^2 .

In order to detect if a variable z_i is assigned (for each solution of $\mathcal{S}'(\mathcal{AC})$ bounded by M) a value different from the original value v_i (that is, if $|y_i| > 0$), a new binary variable δ_i will be defined. The variable δ_i will have value 1 if the value of z_i differs from v_i , 0 otherwise. To express this condition, we add the following constraints to $\mathcal{S}'(\mathcal{AC})$:

$$\begin{cases} y_i \leq M\delta_i \quad \forall i \in \{1, \dots, N\} \\ -M\delta_i \leq y_i \quad \forall i \in \{1, \dots, N\} \\ \delta_i \in \{0, 1\} \quad \forall i \in \{1, \dots, N\} \end{cases} \quad (7.1)$$

The system obtained by assembling $\mathcal{S}'(\mathcal{AC})$ with inequalities (7.1) will be denoted as $\mathcal{S}''(\mathcal{AC})$. For each solution s'' of $\mathcal{S}''(\mathcal{AC})$, the following hold:

- i) for each z_i which is assigned in s'' a value greater than v_i , the variable δ_i is assigned 1 (this is entailed by constraint $y_i \leq M\delta_i$);
- ii) for each z_i which is assigned in s'' a value less than v_i , the variable δ_i is assigned 1 (this is entailed by constraint $-M\delta_i \leq y_i$).

Moreover, for each z_i which is assigned in s'' the same value as v_i (that is, $y_i = 0$), variable δ_i is assigned either 0 or 1.

Obviously each solution of $\mathcal{S}''(\mathcal{AC})$ corresponds to exactly one solution for $\mathcal{S}(\mathcal{AC})$ (or, analogously, for $\mathcal{S}'(\mathcal{AC})$) with the same values for variables z_i , and, vice versa, for each solution of $\mathcal{S}(\mathcal{AC})$ whose variables are bounded by M there is at least one solution of $\mathcal{S}''(\mathcal{AC})$ with the same values for variables z_i . As solutions of $\mathcal{S}(\mathcal{AC})$ correspond to repairs for D , each solution of $\mathcal{S}''(\mathcal{AC})$ corresponds to a repair ρ for D w.r.t. \mathcal{AC} such that, for each update $u = \langle t, A, v \rangle$ in ρ it holds that $|v| \leq M$. Repairs satisfying this property will be said to be *M-bounded repairs*.

In order to consider only the solutions of $\mathcal{S}''(\mathcal{AC})$ where each δ_i is 0 if $y_i = 0$, we consider the following optimization problem $\mathcal{S}^*(\mathcal{AC})$, whose goal is minimizing the sum of the values assigned to the variables $\delta_1, \dots, \delta_N$:

² Observe that the size of M is polynomial in the size of the database, as it is bounded by $\log n + (2 \cdot m + 1) \cdot \log(ma)$.

$$\min \sum_{i=1}^N \delta_i$$

$$\begin{cases} AZ \leq B \\ y_i = z_i - v_i \quad \forall i \in \{1, \dots, N\} \\ y_i - M\delta_i \leq 0 \quad \forall i \in \{1, \dots, N\} \\ -y_i - M\delta_i \leq 0 \quad \forall i \in \{1, \dots, N\} \\ z_i, y_i \in \mathbb{R} \quad \forall i \in I_{\mathbb{R}} \\ z_i, y_i \in \mathbb{Z} \quad \forall i \in I_{\mathbb{Z}} \\ \delta_i \in \{0, 1\} \quad \forall i \in \{1, \dots, N\} \end{cases}$$

where $I_{\mathbb{R}} \subseteq \{1, \dots, N\}$ and $I_{\mathbb{Z}} \subseteq \{1, \dots, N\}$ are the sets of the indexes of the variables z_1, \dots, z_N (and, equivalently, y_1, \dots, y_N) defined on the domains \mathbb{R} and \mathbb{Z} , respectively.

It is straightforward to see that any solution of $\mathcal{S}^*(\mathcal{AC})$ corresponds to an M -bounded repair ρ for D w.r.t. \mathcal{AC} having minimum cardinality w.r.t. all M -bounded repairs for D w.r.t. \mathcal{AC} . If there is a repair for D w.r.t. \mathcal{AC} , then there is an M -bounded *card*-minimal repair ρ^* for D (this follows from Lemma 6.1). This implies that any solution of $\mathcal{S}^*(\mathcal{AC})$ corresponds to a *card*-minimal repair for D w.r.t. \mathcal{AC} .

Basically, the minimum value of the objective function of $\mathcal{S}^*(\mathcal{AC})$ represents the number of atomic updates performed by any *card*-minimal repair, whereas the values of variables $z_1, \dots, z_N, y_1, \dots, y_N, \delta_1, \dots, \delta_N$ corresponding to an optimum solution s^* of $\mathcal{S}^*(\mathcal{AC})$ define the atomic updates performed by the *card*-minimal repair $\rho(s^*)$.

Example 7.1. The optimization problem obtained starting from the database in the Fig. 7.3 and from the set of steady aggregate constraints consisting of 1), 2) and 3) of Example 7.1 is shown in Fig. 7.4. Since it is assumed that the domain of attribute *Value* of relation *CashBudget* is \mathbb{Z} , then $I_{\mathbb{Z}} = \{1, \dots, 20\}$ and $I_{\mathbb{R}} = \emptyset$. The value of the constant M is $20 \cdot (28 \cdot 250)^{2 \cdot 28 + 1}$.

The minimum value of the objective function of this optimization problem is 1 (only $\delta_4 = 1$). This problem admits only one optimum solution where the value of each variable y_1, \dots, y_{20} is 0 except for y_4 that takes value -30 . Clearly, the values of variables z_1, \dots, z_{20} are obtained according to the values of variables y_1, \dots, y_{20} , and the values of binary variables $\delta_1, \dots, \delta_{20}$ are 0 except for δ_4 . The *card*-minimal repair corresponding to this solution is that of Example 7.2. □

7.5 DART Architecture

The DART architecture is shown in Fig. 7.5, where the organization of both the *Acquisition and extraction module* and the *Repairing module* of Fig. 7.2 are described in more detail. In the following we discuss the tasks accomplished by these modules.

$$\begin{cases}
 \min \sum_{i=1}^{20} \delta_i \\
 z_2 + z_3 = z_4 & y_9 = z_9 - 60 \\
 z_5 + z_6 + z_7 = z_8 & y_{10} = z_{10} - 80 \\
 z_{12} + z_{13} = z_{14} & y_{11} = z_{11} - 80 \\
 z_{15} + z_{16} + z_{17} = z_{18} & y_{12} = z_{12} - 100 \\
 z_4 - z_8 = z_9 & y_{13} = z_{13} - 100 \\
 z_{14} - z_{18} = z_{19} & y_{14} = z_{14} - 200 \\
 z_1 - z_9 = z_{10} & y_{15} = z_{15} - 130 \\
 z_{11} - z_{19} = z_{20} & y_{16} = z_{16} - 40 \\
 y_1 = z_1 - 20 & y_{17} = z_{17} - 20 \\
 y_2 = z_2 - 100 & y_{18} = z_{18} - 190 \\
 y_3 = z_3 - 120 & y_{19} = z_{19} - 10 \\
 y_4 = z_4 - 250 & y_{20} = z_{20} - 90 \\
 y_5 = z_5 - 120 & y_i - M\delta_i \leq 0 \quad \forall i \in \{1, \dots, 20\} \\
 y_6 = z_6 - 0 & -y_i - M\delta_i \leq 0 \quad \forall i \in \{1, \dots, 20\} \\
 y_7 = z_7 - 40 & z_i, y_i \in \mathbb{Z} \quad \forall i \in \{1, \dots, 20\} \\
 y_8 = z_8 - 160 & \delta_i \in \{0, 1\} \quad \forall i \in \{1, \dots, 20\}
 \end{cases}$$

Fig. 7.4. MILP-problem instance for the running example

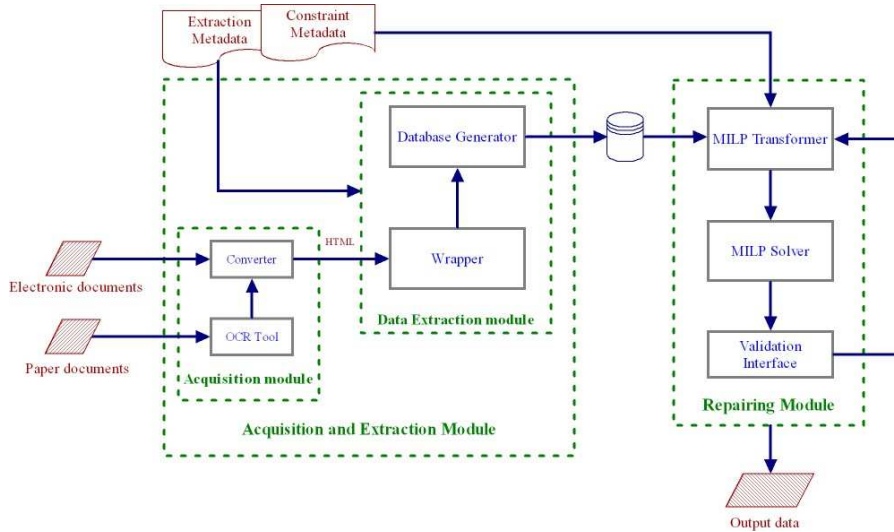


Fig. 7.5. The DART Architecture

7.5.1 Acquisition Module

This module performs the task of acquiring the information contained in the (either electronic or paper) input documents, and represents it into an electronic document whose format is suitable for the extraction phase accomplished by the *Data Extraction Module*. As the current implementation of

DART embeds a wrapper working on HTML documents, input documents which are not already in this format are converted into an HTML document by means of a format-conversion tool (this tool supports the conversion of PDF, MSWord, RTF documents). In particular, paper documents are first digitized and processed by means of an OCR tool (yielding PDF documents) whose output is then processed by the converter.

7.5.2 Data Extraction Module

The *Data extraction module* carries out both the information extraction and the database generation tasks. The former task is accomplished by a wrapping sub-module which takes as input the HTML document generated by the *Acquisition module* as well as a set of extraction metadata providing information on the semantics and the structure of data contained into the input document.

Wrapper

Data to be extracted from the input HTML document are contained into tables whose position inside the document is specified inside the extraction metadata. The information encoded into each table is extracted by evaluating whether its rows match some patterns (namely *row patterns*) defining structure and content of the data to be extracted.

Before explaining how the wrapping sub-module works, we give some details about the set of extraction metadata.

This set contains *domain descriptions*, *row patterns* and *hierarchical relationships*. *Domain descriptions* specify a set of domains and the sets of lexical items that belongs to each domain. For instance, considering the balance sheet analysis context, *Section* and *Subsection* are domains. Some lexical items belonging to the former are “Receipts”, “Disbursements”, “Balance”, whereas some lexical items belonging to the latter are “beginning cash”, “receivables”, “payment of accounts” and “capital expenditure”. In the following we will denote the set of these domains as *Dom*. *Hierarchical relationships* are relations among lexical items belonging to different domains. For instance, the items “beginning cash”, “cash sales”, “receivables” and “total cash receipts” are specializations of “Receipts”. Fig. 7.6 depicts some domains, some lexical items belonging to them and some hierarchical relationships represented by means of arrows.

A *row pattern* specifies the structure and the content of a table row. The structure is given specifying an ordered set of cells. The content of a cell is either a domain belonging to *Dom* or a *standard domain* such as *Integer*, *String*, etc. A row pattern r matches a row r_t of a table in an input document if r and r_t have the same number of cells and if the content of the i -th cell of r_t matches the domain specified into the i -th cell of r . A row pattern contains an *headline* indicating the semantics of the domains specified in the cells.

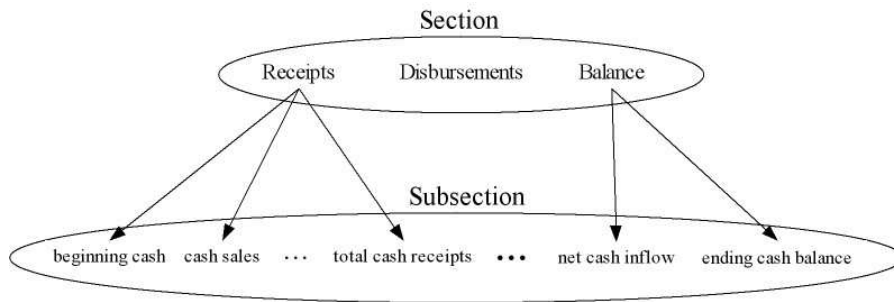


Fig. 7.6. Domains and hierarchical relationships

The headline will be exploited in the database generation task to construct a relation scheme. In a row pattern, hierarchical relationships can be specified among lexical items expected in some cells. For instance, it is possible to require that a lexical item expected in a cell must be a generalization of another lexical item required in another cell.

Example 7.5 Consider the row pattern shown in Fig. 7.7(a). The headline consists of the cells with the dashed border. The row pattern indicates that the rows which must be extracted from the input table consist of 4 cells. In particular, both the first and the last cells specify that a value of type *Integer* is required, and the headline specifies that the first value is interpreted as *Year* and the last as *Value*. The second cell indicates that a lexical item s_1 belonging to the *Section* domain is expected. The third cell imposes a hierarchical relationship, indicated by an arrow. It specifies that a lexical item s_2 belonging to the *Subsection* domain is required, and that s_2 must be specialization of s_1 .

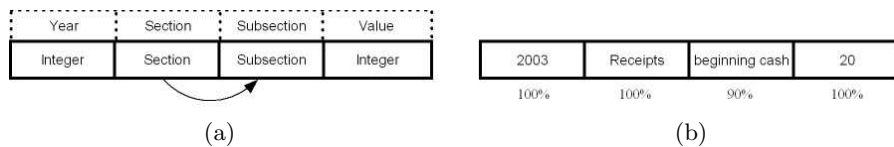


Fig. 7.7. (a) A row pattern (b) A row pattern instance

□

The wrapper takes as input a set of row patterns and the HTML document returned by the acquisition module, and returns a set of *row pattern instances*. A row pattern instance is the result of the matching between a table row and the set of row patterns. First, for each row r_t of the input table, the wrapper identifies the row pattern r that matches r_t at best, i.e. it chooses the row pattern having the most similar structure and the most compatible content

with respect to r_t . After this choice the wrapper constructs the row pattern instance p relative to r .

In more detail, the evaluation of the matching between a table row and a row pattern yields a score representing the *matching degree*. The matching is performed comparing the table cells and the corresponding row pattern cells. The comparison between a row pattern cell and an input table cell yields a cell matching score. The whole row pattern instance is associated with a score obtained by applying a suitable t -norm to all the matching scores of its cells.

Each cell matching score results from “validating” the string s in the table cell w.r.t the domain d specified in the cell of the row pattern. The validation of s w.r.t. d is accomplished by identifying the item s' in d which is the most similar³ to s , and returning the similarity degree between s' and s . Given a string s and a domain d we denote the item in d which is the most similar to s as $msi(d, s)$. The string [resp. the domain] contained in the i -th cell of a document row r_t [resp. row pattern r] will be denoted as $r_t(i)$ [resp. $r(i)$].

For each document row r_t , the row pattern r for which the matching degree is maximum is chosen. Then a row pattern instance p is constructed, where the i -th cell of p contains the item $msi(r(i), r_t(i))$.

Observe that the construction of the row pattern instance is a *form of repair* on the input data. Indeed, incorrect items in the input tables (i.e. items which do not belong to the corresponding domain in the specified row pattern) are transformed into the most similar valid lexical items.

Finally, we obtain a set of row pattern instances such that each document row is mapped on a row pattern instance.

Example 7.6 Consider the document in Fig. 7.1 and the row pattern in Fig. 7.7(a). Assume that a symbol recognition error in non-numerical string occurs, like the recognizing of the item “bgnning cesh” instead of “beginning cash”.

The matching between the first document row and this row pattern returns the row pattern instance in Fig. 7.7(b), where *Integer* in the first cell is bound to “2003”, *Section* to “Receipts”, *Subsection* to “beginning cash” and *Integer* in the last cell is bound to “20”. In Fig. 7.7(b) the matching scores for the cells are also depicted. The third cell score (90%) is lower than the others (100%), since it comes from a non-exact match.

Note that the value “2003” is coded into a multi-row cell of the input table, and it is bound in this row pattern instance since the wrapper considers this value associated to all the document rows which are adjacent to the multi-row cell.

□

³ s' must also satisfy the hierarchical relationships specified in the row pattern.

Database Generator

The *Database generator* sub-module takes as input the set of row pattern instances returned by the wrapper module and returns a database instance D conforming to the database scheme defined in the extraction metadata.

Extraction metadata specify also *classification information* providing classification of lexical items depending on the role they play in aggregation constraints. For instance, in Example 7.1 lexical items in the domain *Subsection* are classified as *detail*, *aggregate* and *derived* items. As explained in Example 7.2, an item is *aggregate* if it is obtained by aggregating items of type *detail* of the same section, whereas a *derived* item is an item whose value can be computed using the values of other items of any type and belonging to any section.

The definition of the database scheme contained in the extraction metadata contains both the definition of the relation scheme (that is, the name of the relations and, for each relation, the names of its attributes) and the correspondence between each relation scheme and the row pattern instances taken as input. For instance in our example the relation scheme specified in the extraction metadata consists of *CashBudget(Year, Section, Subsection, Type, Value)*. Moreover, the extraction metadata contain the specification that attributes *Year, Section, Subsection, Value* correspond to the cells of the row pattern instances described by the same names in the headline, whereas the attribute *Type* is determined by classification information.

Each row pattern instance taken as input is exploited to insert a new tuple in the corresponding relation. For instance, each tuple t in Fig. 7.3 is obtained from a row pattern instance r returned by the wrapper. In particular, the values of the attributes *Year, Section, Subsection, Value* in t are taken from the corresponding cells of the row pattern instance r . Moreover the value of the attribute *Type* is implied by the value of the attribute *Subsection* according to classification information.

7.5.3 Repairing Module

The input of the repairing module is the database D obtained by the data extraction module and a set \mathcal{AC} of steady aggregate constraints implied by the constraint metadata. The repairing module returns a *card*-minimal repair for D w.r.t. \mathcal{AC} . This is accomplished by means two phases: first, the problem of finding a *card*-minimal repair for D w.r.t. \mathcal{AC} is translated into an instance of an MILP problem (as we have shown in Section 7.4), and then such an obtained MILP instance is solved by means of an MILP solver, which is implemented using LINDO API 4.0 (available at www.lindo.com).

Validation Interface

The *Validation Interface* is the component allowing the *operator* to interact with DART. When a document is processed, the *Validation Interface* displays

the repair computed by the *Repairing module* by showing the suggested set of value updates. Then, the operator examines the proposed repair by comparing every updated value with the corresponding source value in the input document. If the operator verifies that the suggested updated values are equal to the corresponding source values, then the repair is accepted and the repaired data is considered as consistent. Otherwise, a new repair is computed by the *Repairing module* according to operator “instructions”. That is, for each suggested update u which has not been accepted by the operator, the operator can specify the actual source value v corresponding to the database item d changed by u . Then an aggregate constraint is added to the set of constraints inputted into the MILP transformer, forcing the value of d to be equal to v . Similarly, accepting an update u on the database item d is translated into an aggregate constraint forcing the value of d to be equal to the value suggested by the repair. After this, a new repair is computed, corresponding to the solution of the new MILP instance obtained by assembling the aggregate constraints resulting from *Constraint Metadata* with those resulting from operator validation. This process goes on until the generated repair is accepted by the operator.

At each iteration, the operator is not requested to validate values which had been already validated in a previous iteration. Moreover, the computation of a repair can be re-started after validating only some of the suggested updates. Every repair is proposed to the operator by displaying its updates in a specific order. That is, an update u_1 is displayed before another update u_2 if the database item d_1 changed by u_1 is involved in a larger number of ground aggregate constraints than the database item d_2 changed by u_2 (i.e. if the variable corresponding to d_1 occurs in the MILP instance in a larger number of inequalities than the variable corresponding to d_2). This ordered displaying is an heuristics which is useful in the case that the operator chooses to re-start the repair computation after a small number of validations, and it aims at finding an acceptable repair in a small number of iterations.

7.6 Discussion

In this chapter we have defined a restricted class of aggregate constraints, namely *steady aggregate constraints*, and we have provided a method for computing a *card*-minimal repair. According to the *card*-minimal semantics, a repaired database D' minimally differs from the original database D if and only if the number of value updates yielding D' is minimum with respect to all other possible repairs (cfr. Section 6.4.2). We have introduced a system architecture, namely *DART* (*Data Acquisition and Repairing Tool*), aiming at supervised acquiring of information encoded into tabular data inside documents with possibly heterogeneous formats. We have shown how the method introduced for computing *card*-minimal repairs can be exploited in the DART

system where data are acquired by means of acquisition tool and information is extracted and transformed by a wrapping system.

There has been a lot of research work related to web information extraction. Specialized information extraction procedures, called *wrappers*, represent an effective solution to capture text contents of interest from a source-native format and encode such contents into a structured format suitable for further application-oriented processing. Web wrappers typically exploit markup-tag and lexical token information to infer the template structuring the contents in a web page [59]. Traditional issues concerning wrapper systems are the development of powerful languages for expressing extraction patterns and the ability of generating these patterns with the lowest human effort [13, 33]. Several systems for generating web wrappers have been recently proposed. We mention here Lixto [13], RoadRunner [33], SCRAP [40, 38], DEByE [58], XWRAP [64], W4F [72]. We point out that the wrapping technique embedded into DART system differs from the state of the art as our approach is mainly focused on data represented into tables with complex structure. The wrapping module embedded in DART can manage tables having “variable” structures, i.e. tables whose cells can span multiple rows and columns, according to no pre-determined scheme (cfr. Section 7.5.2). This is a valuable feature, as all existing wrapping techniques do not work at all or are far from being satisfactory on tabular data without a “rigid” structure.

A framework for computing *card*-minimal repairs on wrongly acquired data has been introduced to drive the data validation process. This framework exploits steady aggregate constraints, defined on the source documents, to check the consistency of the acquired data and to compute a repair. In order to compute a *card*-minimal repair for a database w.r.t a set of steady aggregate constraints, we translate the repair-evaluation problem into an instance of a Mixed-Integer Linear Programming (MILP) problem [48], exploiting the restrictions imposed on steady aggregate constraints w.r.t. general aggregate constraints (cfr. Section 7.4). Although, steady aggregate constraints are a restricted class of (general) aggregate constraints introduced in Chapter 6, we have shown that the complexity results characterizing either the repair existence problem and the consistent query answer problem (provided in Chapter 6) are still valid for the class of steady aggregate constraints (cfr. Section 7.3). Observe that, as the repair-existence problem is *NP*-complete also in presence of steady aggregate constraints, there is no ε -approximation algorithm \mathcal{A} [68] for the computation of a *card*-minimal repair for D , unless $P = NP$. Otherwise, running \mathcal{A} would result in obtaining a possible repair for D (not necessarily a *card*-minimal one) in polynomial time.

Our approach consists in translating the problem of computing a repair into an instance of an MILP problem (cfr. Section 7.4). Thus, standard techniques and optimizations addressing MILP problems can be re-used for computing a repair.

DART is currently being developed. The *Repairing* module has been implemented and preliminary tests show that DART effectively supports the

acquisition of balance data, providing the correct repair of wrongly acquired data in a few iterations in most cases.

Conclusions

Several issues related to the problem of extracting reliable information from inconsistent data have been investigated in literature. The early theoretical approaches to the problem of dealing with incomplete and inconsistent information date back to 80s, but these works mainly focus on issues related to the semantics of incompleteness [53]. The problem of managing inconsistent data in databases was first addressed in [2, 3, 35, 62, 20]. Then, based on the notions of repair and consistent query answer introduced in [4], several works investigated different aspects of the problem of extracting reliable information from inconsistent data. The technique of query-rewritten introduced in [4] was further extended in [45, 46], and in [7, 8, 29] graph theory has been exploited for providing complexity characterization and algorithms for computing consistent answers. Logic programs have been used in [5, 9, 11, 12, 51, 52] for specifying repairs as answer sets of such programs. All the above-cited approaches assume that tuple insertions and deletions are the basic primitives for repairing inconsistent data. In [42, 17, 76, 77, 78, 16] repairs consisting also of value-update operations were considered. Different classes of queries and constraints have been studied in the cited works. But, all works focus on “traditional” forms of constraints such as functional dependencies, inclusion dependencies and denial constraints.

In this work of thesis we have investigated the problem of repairing and extracting reliable information from databases violating *aggregate constraints*, which consist of linear inequalities on aggregate-sum queries issued on values stored in the database. We fill a gap in previous works dealing with inconsistent data, where only traditional forms of constraints were considered. The first work investigating aggregate constraints on numerical data is [71], where the consistency problem of very general forms of aggregation is considered, but no issue related to data-repairing is investigated. In [16] the problem of deciding the existence of a repair in presence of a *non-linear* form of aggregate constraints has been investigated and the authors shown that it is undecidable. Then, they disregard aggregate constraints and focus on the problem of repairing data violating constraints where no form of aggregation is allowed. On

the other hand, in this work of thesis we have shown that the *repair-existence problem* for *linear* form of aggregate constraints is *NP*-complete. This form of aggregate constraints frequently occur in many real-life scenarios where guaranteeing the consistency of numerical data is mandatory.

As in [42, 17], we consider database repairs consisting of sets of value-update operations aiming at re-constructing the correct values of inconsistent data, and such that the values of the key attributes do not change. We investigate two repair-semantics aiming at preserving the information represented in the source data as much as possible, namely the *set*-minimal semantics and *card*-minimal semantics. They correspond to different repairing strategies which turn out to be well-suited for different application scenarios. We have characterized the data-complexity of the *minimal-repair checking problem* and the *consistent query answer problem* under both these semantics in presence of aggregate constraints.

We have provided a method for computing *card*-minimal repairs for databases which are inconsistent with respect to *steady aggregate constraints*, a restricted but expressive class of aggregate constraints. Using steady aggregate constraints the loss in expressiveness is not dramatic, as they suffice to express relevant integrity constraints in many real-life scenarios. Moreover, we have shown that all the complexity results valid for aggregate constraints still hold for steady aggregate constraints.

Our approach for computing repairs consists in translating an instance of the problem of computing a *card*-minimal repair into an instance of a Mixed-Integer Linear Programming (MILP) problem. This method allows us re-using of standard techniques addressing MILP problems for efficiently computing a repair for numerical data.

We have introduced a system architecture, namely DART (*Data Acquisition and Repairing Tool*), aiming at supervised acquiring of information encoded into tabular data inside documents with possibly heterogeneous formats. We have shown how the method introduced for computing *card*-minimal repairs can be exploited in the DART system where data are acquired by means of acquisition tool and information is extracted and transformed by a wrapping system.

Regarding our repairing framework, some related issues remain still open. For instance, it would be interesting the identification of other decidable cases of aggregate constraints, when more expressive forms of constraint are adopted. Basically, for our form of aggregate constraints, decidability results from the linear nature of the considered constraints, where products between *measure attributes* in attribute expressions of the aggregation functions were not allowed. Another interesting issue is the characterization of the repair-existence and the consistent query answer problem when both aggregate constraints and traditional forms of integrity constraints are violated (as it has been done in [23, 29] for data violating different forms of classical integrity constraints). Moreover, since we deal with numerical data it would be interesting the characterization of the consistent query answer problem under

range-semantics and *aggregate queries* [7, 46]. Further, it would be interesting the extension of our repairing framework with the introduction of *weak* constraints [42] in order to prefer specific repairs among the minimal ones. Specifically, among minimal repairs, we could prefer repairs that satisfies the maximum number of weak constraints, each of them specifying additional conditions such as degrees of agreement on historical data. Moreover, it would be interesting the design of efficient algorithms for computing consistent answers by exploiting a method similar to that used for computing *card*-minimal repairs.

Recently, *incremental* repair semantics [66] has been investigated. In this context it is assumed that the database is already consistent before updates are executed and incremental version of consistent query answer has been provided. Moreover, a study of the problem of computing consistent answers under normalization conditions has been reported in [79] in presence of functional dependencies. It would be interesting a study of these aspects of the problem of computing consistent answers in presence of the several forms of traditional integrity constraints, and also in presence of aggregate constraints.

References

1. Abiteboul, S., Hull, R., Vianu, V., *Foundations of Databases*, Addison-Wesley, 1995.
2. Agarwal, S., *Flexible Relation: A Model for Data in Distributed, Autonomous and Heterogeneous Databases*, Ph.D. Thesis, Department of Electrical Engineering, Stanford University, 1992.
3. Agarwal, S., Keller, A. M., Wiederhold, G., Saraswat, K., Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases, *Proc. International Conference on Data Engineering (ICDE)*, pages 495–504, 1995.
4. Arenas, M., Bertossi, L. E., Chomicki, J., Consistent Query Answers in Inconsistent Databases, *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 68–79, 1999.
5. Arenas, M., Bertossi, L. E., Chomicki, J., Specifying and Querying Database Repairs using Logic Programs with Exceptions, *Proc. International Conference on Flexible Query Answering Systems (FQAS)*, pages 27–41, 2000.
6. Arenas, M., Bertossi, L. E., Kifer, M., Applications of Annotated Predicate Calculus to Querying Inconsistent Databases, *Proc. International Conference on Computational Logic (CL)*, pages 926–941, 2000.
7. Arenas, M., Bertossi, L. E., Chomicki, J., Scalar Aggregation in FD-Inconsistent Databases, *Proc. International Conference on Database Theory (ICDT)*, pages 39–53, 2001.
8. Arenas, M., Bertossi, L. E., Chomicki, J., He, X., Raghavan, V., Spinrad, J., Scalar aggregation in inconsistent databases, *Theoretical Computer Science*, Vol. 3(296), pages 405–434, 2003.
9. Arenas, M., Bertossi, L. E., Chomicki, J., Answer Sets for Consistent Query Answering in Inconsistent Databases, *Theory and practice of logic programming*, Vol. 3(4-5), pages 393–424, 2003.
10. Baral, C., Gelfond, G., Logic Programming and Knowledge Representation, *Journal of Logic Programming*, Vol. 19/20, pages 73–148, 1994.
11. Barceló, P., Bertossi, L. E., Repairing Databases with Annotated Predicate Logic, *Proc. International Workshop on Non-Monotonic Reasoning (NMR)*, pages 160–170, 2002.

12. Barceló, P., Bertossi, L. E., Logic Programs for Querying Inconsistent Databases, *Proc. Practical Aspects of Declarative Languages (PADL)*, pages 208–222, 2003.
13. Baumgartner, R., Flesca, S., Gottlob, G., Visual Web Information Extraction with Lixto, *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 119–128, 2001.
14. Beeri, C. and Vardi, M. Y., A Proof Procedure for Data Dependencies, *Journal of the ACM*, Vol. 31(4), pages 718–741, 1984.
15. Bertossi, L. E., Chomicki J., Query Answering in Inconsistent Databases, In *Logics for Emerging Applications of Databases*, pages 43–83, 2003.
16. Bertossi, L. E., Bravo, L., Franconi, E., Lopatenko, A., Complexity and Approximation of Fixing Numerical Attributes in Databases Under Integrity Constraints, *Proc. International Symposium on Database Programming Languages (DBPL)* pages 262–278 2005.
17. Bohannon, P., Flaster, M., Fan, W., Rastogi, R., A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification, *Proc. ACM SIGMOD International Conference on Management of Data*, pages 143–154, 2005.
18. Borosh, I., Treybig, L. B., Bounds on Positive Integral Solutions of Linear Diophantine Equations, *American Mathematical Society*, Vol. 55(2), pages 299–304, 1976.
19. Bry, F., A Compositional Semantics for Logic Programs and Deductive Databases, *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 453–467, 1996.
20. Bry, F., Query Answering in Information Systems with Integrity Constraints, *Proc. Working Conference on Integrity and Control in Information Systems (IICIS)*, pages 113–130, 1997.
21. Buccafurri, F., Leone N., Rullo P., Enhancing Disjunctive Datalog by Constraints, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12(5), pages 845–860, 2000.
22. Cadoli, M., Donini, F. M., Liberatore, P., Schaerf, M., Feasibility and Unfeasibility of Off-Line Processing, *Proc. Israel Symposium on Theory of Computing Systems (ISTCS)*, pages 100–109, 1996.
23. Calí, A., Lembo, D., Rosati, R., On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases, *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 260–271, 2003.
24. Celle, A., Bertossi, L., Querying Inconsistent Databases: Algorithms and Implementation, *Proc. International Conference on Computational Logic (CL)*, pages 942–956, 2000.
25. Chakravarthy, U. S., Grant, J., Minker, J., Logic-Based Approach to Semantic Query Optimization, *ACM Transactions on Database Systems*, Vol. 15(2), pages 162–207, 1990.
26. Chandra, A. K., Merlin, P. M., Optimal Implementation of Conjunctive Queries in Relational Data Bases, *Symposium on the Theory of Computing (STOC)*, pages 77–90, 1977.

27. Chomicki, J., Marcinkowski, J., Staworko, S., Computing Consistent Query Answers Using Conflict Hypergraphs, *Proc. International Conference on Information and Knowledge Management (CIKM)*, pages 417–426, 2004.
28. Chomicki, J., Marcinkowski, J., Staworko, S., Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. *Proc. International Conference on Extending Database Technology (EDBT)*, System demo, pages 841–844 2004.
29. Chomicki, J., Marcinkowski, J., Minimal-Change Integrity Maintenance Using Tuple Deletions, *Information and Computation*, Vol. 197(1-2), pages 90–121, 2005.
30. Chomicki, J., Marcinkowski, J., On the Computational Complexity of Minimal-Change Integrity Maintenance in Relational Databases, In *Inconsistency Tolerance*, pages 119–150, 2005.
31. Chomicki, J., Consistent Query Answering: Opportunities and Limitations, *Proc. DEXA Workshop on Logical Aspects and Applications of Integrity Constraints (LAAIC)*, pages 527–531, 2006.
32. Clark K. L., Negation as Failure, *Symposium on Logic and Data Bases*, pages 293–322, 1977.
33. Crescenzi, V., Mecca, G., Merialdo, P., RoadRunner: Towards Automatic Data Extraction from Large Web Sites, *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 109–118, 2001.
34. Dalal, M., Investigations Into a Theory of Knowledge Base Revision, *Proc. National Conference of the American Association for Artificial Intelligence (AAAI)*, pages 475–479, 1988.
35. Dung, P. M., Integrating Data from Possibly Inconsistent Databases, *Proc. International Conference on Cooperative Information Systems (CoopIS)*, pages 58–65, 1996.
36. Eiter, T., Gottlob, G., On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactual, *Artificial Intelligence*, Vol. 57(2-3), pages 227–270, 1992.
37. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello F., The Knowledge Representation System DLV: Progress Report, Comparisons and Benchmarks, *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 406–417, 1998.
38. Fazzinga, B., Flesca, S., Tagarelli, A., Learning Robust Web Wrappers, *Proc. International Conference on Database and Expert Systems Applications (DEXA)* pages 736–745, 2005.
39. Fazzinga, B., Flesca, S., Furfaro, F., Parisi, F., DART: A Data Acquisition and Repairing Tool, *Proc. EDBT Workshop on Inconsistency and Incompleteness in Databases (IIDB)*, pages 297–317, 2006.
40. Flesca, S., Tagarelli, A., Schema-Based Web Wrapping, *Proc. International Conference on Conceptual Modeling (ER)*, pages 286–299, 2004.
41. Flesca, S., Furfaro, F., Parisi, F., Consistent Query Answer on Numerical Databases under Aggregate Constraint, *Proc. International Symposium on Database Programming Languages (DBPL)*, pages 279–294, 2005.
42. Franconi, E., Palma, A. L., Leone, N., Perri, S., Scarcello, F., Census Data Repair: a Challenging Application of Disjunctive Logic Programming, *Proc. Interna-*

- tional Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 561-578, 2001.
43. Fremuth-Paeger, C., Jungnickel, D., Balanced Network Flows. I. A Unifying Framework for Design and Analysis of Matching Algorithms, *Networks*, Vol. 33(1), pages 1-28, 1999.
 44. Fuxman, A., Miller, R. J., Towards Inconsistency Management in Data Integration Systems. *Proc. IJCAI Workshop on Information Integration on the Web (IIWeb)*, pages 143-148, 2003.
 45. Fuxman, A., Miller, R. J., First-Order Query Rewriting for Inconsistent Databases, *Proc. International Conference on Database Theory (ICDT)*, pages 337-351, 2005.
 46. Fuxman, A., Miller, R. J., ConQuer: Efficient Management of Inconsistent Databases, *Proc. ACM SIGMOD International Conference on Management of Data*, pages 155-166, 2005.
 47. Fuxman, A., Miller, R. J., ConQuer: A System for Efficient Querying Over Inconsistent Databases, *Proc. International Conference on Very Large Data Bases (VLDB)*, System demo, pages 1354-1357, 2005.
 48. Gass, S. I., *Linear Programming Methods and Applications*, McGrawHill, 1985.
 49. Gelfond, M., Lifschitz, V., The Stable Model Semantics for Logic Programming, *Proc. International Conference and Symposium on Logic Programming (ICLP/SLP)*, Vol. 2, pages 1070-1080, 1988.
 50. Gelfond, M., Lifschitz, V., Classical Negation in Logic Programs and Disjunctive Databases *New Generation Computing*, Vol. 9(3-4), pages 365-386, 1991.
 51. Greco, S., Zumpano, E., Querying Inconsistent Databases, *Proc. International Conference on Logic for Programming and Automated Reasoning (LPAR)*, pages 308-325, 2000.
 52. Greco, G., Greco, S., Zumpano, E., A Logical Framework for Querying and Repairing Inconsistent Databases, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15(6), pages 1389-1408, 2003.
 53. Imielinski, T., Lipski, W., Incomplete Information in Relational Databases, *Journal of the ACM*, Vol. 31(4), pages 761-791, 1984.
 54. Johnson, D. S., Klug, A. C., Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies, *Journal of Computer and System Sciences*, Vol. 28(1), pages 167-189, 1984.
 55. Johnson, D. S., A Catalog of Complexity Classes, In *Handbook of Theoretical Computer Science*, Vol. A, pages 67-161, 1990.
 56. Kifer, M., Lozinskii, E. L., A Logic for Reasoning with Inconsistency, *Journal of Automated Reasoning*, Vol. 9(2), pages 179-215, 1992.
 57. Kowalski, R. A., Sadri, F., Logic Programs with Exceptions, *New Generation Computing*, Vol. 9(3-4), pages 387-400, 1991.
 58. Laender, A. H. F., Ribeiro-Neto, B. A., da Silva, A. S., DEByE - Data Extraction By Example, *Data and Knowledge Engineering*, Vol. 40(2), pages 121-154, 2002.
 59. Laender, A. H. F., Ribeiro-Neto, B. A., da Silva, A. S., Teixeira, J. S., A brief Survey of Web Data Extraction Tools, *SIGMOD Record*, Vol. 31(2), pages 84-93, 2002.

60. Lenzerini, M., Data Integration: A Theoretical Perspective, *Proc. Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.
61. Libkin, L., Wong, L., On Representation and Querying Incomplete Information in Databases with Bags, *Information Processing Letters*, Vol 56(4), pages 209–214, 1995.
62. Lin, J., Mendelzon, A. O., Merging Databases Under Constraints, *International Journal of Cooperative Information Systems*, Vol. 7(1), pages 55–76, 1998.
63. Lin, J., Mendelzon, A. O., Knowledge Base Merging by Majority, In *Dynamic Worlds: From the Frame Problem to Knowledge Management*, 1999.
64. Liu, L., Pu, C., Han, W., XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources, *Proc. International Conference on Data Engineering (ICDE)*, pages 611–621, 2000.
65. Lopatenko, A., Bertossi, L. E., Consistent Query Answering By Minimal-Size Repairs, *Proc. DEXA Workshop on Logical Aspects and Applications of Integrity Constraints (LAAIC)*, pages 558–562, 2006.
66. Lopatenko, A., Bertossi, L. E., Complexity of Consistent Query Answering in Databases under Cardinality-Based and Incremental Repair Semantics, Technical Report arXiv:cs.DB/0604002 v1. Posted April 2, 2006.
67. Papadimitriou, C. H., On the complexity of integer programming, *Journal of the ACM*, Vol. 28(4), pages 765–768, 1981.
68. Papadimitriou, C. H., *Computational Complexity*, Addison-Wesley, 1994.
69. Przymusiński T., Stable Semantics for Disjunctive Programs, *New generation computing* Vol. 9(3-4), pages 401–424, 1991.
70. Reiter R., On Closed World Data Bases *Symposium on Logic and Data Bases*, pages 55–76, 1977.
71. Ross, K. A., Srivastava, D., Stuckey, P. J., Sudarshan, S., Foundations of Aggregation Constraints, *Theoretical Computer Science*, Vol. 193(1-2), pages 149–179, 1998.
72. Sahuguet, A., Azavant, F., Building Intelligent Web Applications Using Lightweight Wrappers, *Data and Knowledge Engineering*, Vol. 36(3), pages 283–316, 2001.
73. Sakama, C., Inoue, K., Prioritized Logic Programming and Its Application to Commonsense Reasoning, *Artificial Intelligence*, Vol. 123(1-2), pages 185–222, 2000.
74. Ullman., J., *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Vol. I, 1988.
75. Wang, X., You, J. H., Yuan L. Y., Nonmonotonic Reasoning by Monotonic Inference with Priority Constraints, *Proc. International Workshop on Nonmonotonic Extensions of Logic Programming (NMELP)*, pages 91–109, 1996.
76. Wijsen, J., Condensed Representation of Database Repairs for Consistent Query Answering, *Proc. International Conference on Database Theory (ICDT)*, pages 378–393, 2003.
77. Wijsen, J., Making More Out of an Inconsistent Database, *Proc. International Conference on Advances in Databases and Information Systems (ADBIS)*, pages 291–305, 2004.

78. Wijsen, J., Database Repairing Using Updates, *ACM Transactions on Database Systems*, Vol. 30(3), pages 722–768, 2005.
79. Wijsen, J., Project-Join-Repair: An Approach to Consistent Query Answering Under Functional Dependencies, *Proc. International Conference on Flexible Query Answering Systems (FQAS)*, pages 1–12, 2006.
80. Zang, Y., Foo, N., Answer Sets for Prioritized Logic Programs, *Proc. International Logic Programming Symposium (ILPS)*, pages 69–83, 1997.
81. Zaniolo, C., Database Relations with Null Values, *Journal of Computer and System Sciences*, Vol. 28(1), pages 142–166, 1984.