Jan Chomicki, Gunter Saake,
Ron van der Meyden

Logics for Emerging Applications of Databases

Springer

Berlin Heidelberg New York
Barcelona Budapest Hong Kong
London Milan Paris
Santa Clara Singapore Tokyo

Contents

Query Answering in Inconsistent Databases

Leopoldo Bertossi[1] and Jan Chomicki[2]

[1] School of Computer Science, Carleton University, Ottawa, Canada, bertossi@scs.carleton.ca
[2] Dept. of Computer Science and Engineering, University at Buffalo, State University of New York, Buffalo, NY, chomicki@cse.buffalo.edu

Abstract. In this chapter, we summarize the research on querying inconsistent databases we have been conducting over the last five years. The formal framework we have used is based on two concepts: *repair* and *consistent query answer*. We describe different approaches to the issue of computing consistent query answers: query transformation, logic programming, inference in annotated logics, and specialized algorithms. We also characterize the computational complexity of this problem. Finally, we discuss related research in artificial intelligence, databases, and logic programming.

1 Introduction

In this chapter, we address the issue of obtaining *consistent* information from *inconsistent* databases – databases that violate given integrity constraints. Our basic assumption departs from everyday practice of database management systems. Typically, a database management system checks the satisfaction of integrity constraints and backs out those updates that violate them. However, present-day database applications have to consider a variety of scenarios in which data is not necessarily consistent:

Integration of autonomous data sources. The sources may separately satisfy the constraints, but when they are integrated together the constraints may stop to hold. For instance, consider different, conflicting addresses for the same person in a taxpayer and a voter registration databases. Each of those databases separately satisfies the functional dependency that associates a single address with each person and yet together they violate this dependency. Moreover, since the sources are autonomous they can not be simply fixed to satisfy the dependency by removing one of the conflicting tuples.

Unenforced integrity constraints. Even though integrity constraints capture an important part of the semantics of a given application, they may still fail to be enforced for a variety of reasons. A data source may be a legacy system that does not support the notion of integrity checking altogether. Or, integrity checking may be too costly (this is often the reason for dropping some integrity constraints from a database schema). Finally, the DBMS itself may support only a limited class of constraints.

Temporary inconsistencies. It may often be the case that the consistency of a database is only temporarily violated and further updates or transactions

are expected to restore it. This phenomenon is becoming more and more common, as the databases are increasingly involved in a variety of long-running activities or *workflows*.

Conflict resolution. Removing tuples from a database to restore consistency leads to information loss, which may be undesirable. For example, one may want to keep multiple addresses for a person if it is not clear which is the correct one. In general, the process of conflict resolution may be complex, costly, and non-deterministic. In real-time decision-making applications, there may not be enough time to resolve all conflicts relevant to a query.

To formalize the notion of consistent information obtained from a (possibly inconsistent) database in response to a user query, we propose the notion of a *consistent query answer*. A consistent answer is, intuitively, true regardless of the way the database is fixed to remove constraint violations. Thus answer consistency serves as an indication of its reliability. The different ways of fixing an inconsistent database are formalized using the notion of a *repair*. A repair is another database that is consistent and minimally differs from the original database.

We summarize the results obtained so far in this area by ourselves and our collaborators. We have studied consistent query answers for first-order and scalar aggregation queries. We have also considered the specification of repairs using logic-based formalisms. We relate our results to similar work undertaken in knowledge representation and logic programming, databases, and philosophical logic. It should be pointed out that we are studying a very specific instance of the logical inconsistency problem: the case where the data is inconsistent with the integrity constraints. We do not address the issue of how to deal with inconsistent sets of formulas in general. In standard relational databases negative information is represented implicitly (through the Closed World Assumption) and inconsistencies appear only in the presence of integrity constraints.

The trivialization of classical logical inference in the presence of an inconsistency is less of a problem in the database context, since database systems typically do not support full-fledged first-order inference. It is more important to be able to distinguish which query answers are affected by the inconsistency and which are not.

This chapter is structured as follows. In Section 2, we define the notions of repair and consistent query answer (CQA) in the context of first-order queries. In Section 3, we present a corresponding computational methodology based on query transformation. In Section 4, we show how to declaratively specify database repairs using logic programming and annotated logics. In Section 5, we discuss computational complexity issues. In Section 6, we show that in the context of aggregation queries the definition of CQAs has to be slightly modified and we discuss the corresponding computational mechanisms. In Section 7, we discuss other, related approaches to handling inconsistent information. In Section 8, we present open problems.

2 Consistent Query Answers

Our basic assumption is that an inconsistent database is not necessarily going to be repaired in a way that fully restores its consistency. Therefore, if such a database is to be queried, we have to distinguish between the information in the database that participates in integrity violations, and one that does not. Typically, only a small part of a database will be inconsistent.

We need to make precise the notion of "consistent" (or "correct") information in an inconsistent database. More specifically, our problem consists of:

1. giving a precise definition of a *consistent answer* to a query in an inconsistent database,
2. finding *computational mechanisms* for obtaining consistent information from an inconsistent database, and
3. studying the *computational complexity* of this problem.

Example 1. Consider the following relational database instance r:

Employee	*Name*	*Salary*
	$J.Page$	5000
	$J.Page$	8000
	$V.Smith$	3000
	$M.Stowe$	7000

The instance r violates the functional dependency $f_1 : Name \rightarrow Salary$ through the first two tuples. This is an inconsistent database. Nevertheless, there is still some "consistent" information in it. For example, only the first two tuples participate in the integrity violation. In order to characterize the consistent information, we notice that there are two possible ways to repair the database in a minimal way if only deletions and insertions of whole tuples are allowed. They give rise to two different repairs:

Employee1	*Name*	*Salary*
	$J.Page$	5000
	$V.Smith$	3000
	$M.Stowe$	7000

Employee2	*Name*	*Salary*
	$J.Page$	8000
	$V.Smith$	3000
	$M.Stowe$	7000

We can see that certain information e.g., $(M.Stowe, 7000)$, persists in both repairs, since it does not participate in the violation of the FD f_1. On the other hand, some information, e.g. $(J.Page, 8000)$, does not persist in all repairs, because it participates in the violation of f_1.

There are other pieces of information that can be found in both repairs, e.g. we know that there is an employee with name *J. Page*. Such information cannot be obtained if we simply discard the tuples participating in the violation.

□

In the following we assume we have a fixed relational database schema R consisting of a finite set of relations. We also have two fixed, disjoint infinite database domains: D (uninterpreted constants) and N (numbers). We assume that elements of the domains with different names are different. The database instances can be seen as finite, first order structures over the given schema, that share the domains D and N. Every attribute in every relation is typed, thus all the instances of R can contain only elements either of D or of N in a single attribute. Since each instance is finite, it has a finite active domain which is a subset of $D \cup N$. As usual, we allow the standard built-in predicates over N $(=, \neq, <, >, \leq, \geq)$ that have infinite, fixed extensions. The domain D has only equality as a built-in predicate. With all these elements we can build a first-order language $\mathcal{L}$.

2.1 Integrity constraints

Integrity constraints are typed, closed first-order $\mathcal{L}$-formulas. We assume that we are dealing with a single set of integrity constraints IC which is consistent as a set of logical formulas. In the sequel we will denote relation symbols by $P_1, \ldots, P_m$, tuples of variables and constants by $\bar{x}_1, \ldots, \bar{x}_m$, and a quantifier-free formula referring to built-in predicates only by ϕ. We also represent a ground tuple $\bar{a}$ in a relation P as the fact $P(\bar{a})$.

Practically important integrity constraints (called simply *dependencies* in [1, chapter 10]) can be expressed as $\mathcal{L}$-sentences of the form

$$\forall \bar{x}\ \exists \bar{y}.\ [\bigvee_{i=1}^{m} P_i(\bar{x}_i)\ \vee\ \bigvee_{i=m+1}^{n} \neg P_i(\bar{x}_i)\ \vee\ \phi(\bar{x}_1, \ldots, \bar{x}_n)], \tag{1}$$

where $\bar{x}_i \subseteq \bar{x} \cup \bar{y}$, $i = 1, \ldots, n$.

In this chapter we discuss the following classes of integrity constraints that are special cases of (1):

1. *Universal integrity constraints*: $\mathcal{L}$-sentences

$$\forall \bar{x}_1, \ldots, \bar{x}_n.\ [\bigvee_{i=1}^{m} P_i(\bar{x}_i)\ \vee\ \bigvee_{i=m+1}^{n} \neg P_i(\bar{x}_i)\ \vee\ \phi(\bar{x}_1, \ldots, \bar{x}_n)].$$

2. *Denial constraints*: $\mathcal{L}$-sentences

$$\forall \bar{x}_1, \ldots, \bar{x}_m.\ [\bigvee_{i=1}^{m} \neg P_i(\bar{x}_i)\ \vee\ \phi(\bar{x}_1, \ldots, \bar{x}_m)].$$

   They are a special case of universal constraints.
3. *Binary constraints*: universal constraints with at most two occurrences of database relations.

4. *Functional dependencies (FDs)*: $\mathcal{L}$-sentences

$$\forall \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4 \bar{x}_5.\ [\neg P(\bar{x}_1, \bar{x}_2, \bar{x}_4)\ \vee\ \neg P(\bar{x}_1, \bar{x}_3, \bar{x}_5)\ \vee\ \bar{x}_2 = \bar{x}_3].$$

   They are a special case of binary denial constraints. A more familiar formulation of the above FD is $X \rightarrow Y$ where X is the set of attributes of P corresponding to $\bar{x}_1$ and Y the set of attributes of P corresponding to $\bar{x}_2$ (and $\bar{x}_3$).
5. *Referential integrity constraints*, also called *inclusion dependencies (INDs)*: $\mathcal{L}$-sentences

$$\forall \bar{x}\ \exists \bar{x}_3.\ [\neg Q(\bar{x}_1)\ \vee\ P(\bar{x}_2, \bar{x}_3)],$$

   where the $\bar{x}_i$ are sequences of distinct variables with $\bar{x}_2$ contained in $\bar{x}_1$, and P, Q database relations. Again, this is often written as $Q[Y] \subseteq P[X]$ where X (resp. Y) is the set of attributes of P (resp. Q) corresponding to $\bar{x}_2$. If P and Q are clear from the context, we omit them and write the dependency simply as $Y \subseteq X$. If an IND can be written without any existential quantifiers, then it is called *full*.

Denial constraints, in particular FDs, and INDs are the most common integrity constraints occurring in database practice. In fact, commercial systems typically restrict FDs to key dependencies and INDs to foreign key constraints.

Given a set of FDs and INDs IC and a relation P with attributes U, a *key* of P is a minimal set of attributes X of P such that IC entails the FD $X \rightarrow U$. In that case, we say that each FD $X \rightarrow Y \in IC$ is a *key* dependency and each IND $Q[Y] \subseteq P[X] \in IC$ is a *foreign key constraint*. If, additionally, X is the primary key of P, then both kinds of dependencies are termed *primary*.

We have seen an FD in Example 1. FDs and INDs are also present in Example 4. Below we show some examples of denial constraints.

Example 2. Consider the relation *Emp* with attributes *Name*, *Salary*, and *Manager*, with *Name* being the primary key. The constraint that *no employee can have a salary greater that that of her manager* is a denial constraint:

$$\forall n, s, m, s', m'.\ [\neg Emp(n, s, m) \vee \neg Emp(m, s', m') \vee s \leq s'].$$

Similarly, single-tuple constraints (`CHECK` constraints in SQL2) are a special case of denial constraints. For example, the constraint that *no employee can have a salary over $200000* is expressed as:

$$\forall n, s, m.[\neg Emp(n, s, m) \vee s \leq 200000].$$

□

Definition 1. Given a database instance r of R and a set of integrity constraints IC, we say that r is *consistent* if $r \models IC$ in the standard model-theoretic sense; *inconsistent* otherwise. □

Reiter [87] characterized relational databases as first-order theories by axiomatizing the unique names, domain closure, and closed world assumptions. Each such a theory is categorical in the sense that it admits the original database, seen as a first-order structure, as its only model. In consequence, satisfaction in a model can be replaced by first-order logical entailment. In this context, a database is consistent with respect to a set of integrity constraints if it entails (as a theory) the set of integrity constraints. There is an alternative notion of database consistency [90]: a database is consistent if its union (as a theory consisting of the atoms in the database) with the set of integrity constraints is consistent in the usual logical sense. All the three notions of consistent relational database, namely the two just presented and Definition 1, turn out to be equivalent for relational databases, but may differ for "open" knowledge bases (see [88,89] for a discussion).

Example 3. Consider a binary relation $P(AB)$ and a functional dependency $A \rightarrow B$. An instance p of P consisting of two tuples (a, b) and (a, c) is inconsistent according to Definition 1. The set of formulas Γ_p defined as:

$$\Gamma_p = \{P(a, b), P(a, c), b \neq c, \forall x, y, z.\ [\neg P(x, y)\ \vee\ \neg P(x, z)\ \vee\ y = z]\}$$

is inconsistent in the standard logic sense. □

2.2 Repairs

Given a database instance r, the *set* $\Sigma(r)$ *of facts* of r is the set of ground atomic formulas $\{P(\bar{a}) \mid r \models P(\bar{a})\}$, where P is a relation name and $\bar{a}$ a ground tuple. The *distance* $\Delta(r, r')$ between data-base instances r and r' is defined as the symmetric difference of r and r':

$$\Delta(r, r')\ =\ (\Sigma(r) - \Sigma(r')\ \cup\ (\Sigma(r') - \Sigma(r)).$$

Definition 2. [3] A database instance r' is a *repair* of a database instance r w.r.t. a set of integrity constraints IC if

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

□

We note that for denial constraints all the repairs of an instance r are subsets of r (see Example 1). However, for more general constraints repairs may contain tuples that do not belong to r. For instance, removing violations of referential integrity constraints can be done not only by deleting but also by inserting tuples.

Example 4. Consider a database with two relations *Personnel(SSN,Name)* and *Manager(SSN)*. There are functional dependencies $SSN \rightarrow Name$ and $Name \rightarrow SSN$, and an inclusion dependency

$$Manager[SSN] \subseteq Personnel[SSN].$$

The relations have the following instances:

Personnel	*SSN*	*Name*
	123456789	*Smith*
	555555555	*Jones*
	555555555	*Smith*

Manager	*SSN*
	123456789
	555555555

The instances do not violate the IND but violate both FDs. If we consider only the FDs, there are two repairs: one obtained by removing the third tuple from *Personnel*, and the other by removing the first two tuples from the same relation. However, the second repair violates the IND. This can be fixed by removing the first tuple from *Manager*. So if we consider all the constraints, there are two repairs obtained by deletion:

Personnel	*SSN*	*Name*
	123456789	*Smith*
	555555555	*Jones*

Manager	*SSN*
	123456789
	555555555

and

Personnel	*SSN*	*Name*
	555555555	*Smith*

Manager	*SSN*
	555555555

Additionally, there are infinitely many repairs, obtained by a combination of deletions and insertions, of the form:

Personnel	*SSN*	*Name*
	123456789	c
	555555555	*Smith*

Manager	*SSN*
	123456789
	555555555

where c is an arbitrary string different from *Smith*. □

Definition 2 reflects the assumption that the information in the database may be not only incorrect but also *incomplete*. This assumption is warranted

in some information integration approaches [72]. On the other hand, restricting repairs to be subsets of the original database (as in [28]) is based on the assumption that the information in the database is *complete*, although not necessarily correct. That assumption seems appropriate in the context of data warehousing where dirty data coming from many sources is cleaned in order to be used as a part of the warehouse itself.

Another variation of the notion of repair assumes a different notion of minimality: instead of minimizing the symmetric difference, we may minimize its *cardinality*. We discuss this issue in Section 7. Still another dimension of the repair concept was recently introduced by Wijsen [95] who proposed repairs obtained by modifying selected tuple components.

2.3 Queries and consistent query answers

Queries are formulas over the same language $\mathcal{L}$ as the integrity constraints. A query is *closed* (or a *sentence*) if it has no free variables. A closed query without quantifiers is also called *ground.*. *Conjunctive queries* [26,1] are queries of the form

$$\exists \bar{x}_1, \ldots \bar{x}_m.\ [P_1(\bar{x}_1)\ \wedge\ \cdots\ \wedge\ P_m(\bar{x}_m)\ \wedge\ \phi(\bar{x}_1, \ldots, \bar{x}_m)]$$

where $\phi(\bar{x}_1, \ldots, \bar{x}_m)$ is a conjunction of built-in predicate atoms. If a conjunctive query has no repeated relation symbols, it is called *simple*.

The following definition is standard:

Definition 3. A tuple $\bar{t}$ is an *answer* to a query $Q(\bar{x})$ in r iff $r \models Q(\bar{t})$, i.e., the formula Q with $\bar{x}$ replaced by $\bar{t}$ is true in r. □

Given a query $Q(\bar{x})$ to an instance r, we want as *consistent* answers those tuples that are unaffected by the violations of the integrity constraints present in r.

Definition 4. [3] A tuple $\bar{t}$ is a *consistent answer* (CQA) to a query $Q(\bar{x})$ in a database instance r w.r.t. a set of integrity constraints IC iff $\bar{t}$ is an answer to the query $Q(\bar{x})$ in every repair r' of r w.r.t. IC. An $\mathcal{L}$-sentence Q is *consistently true* in r w.r.t. IC if it is true in every repair of r w.r.t IC. In symbols:

$$r \models_{IC} Q(\bar{t})\ \equiv\ r' \models Q(\bar{t})\ \text{ for every repair } r' \text{ of } r \text{ w.r.t. } IC.$$

□

Example 5. (Example 1 continued) It holds:

1. $r \models_{\{f_1\}}\ Employee(M.Stowe, 7000)$
2. $r \models_{\{f_1\}}\ (Employee(J.Page, 5000)\ \vee\ Employee(J.Page, 8000))$

3. $r \models_{\{f_1\}} \exists x.\ [Employee(J.Page, x)]$

□

Notice that through Definition 4 our approach leads to a stronger notion of inference from inconsistent databases than an approach based on simply discarding conflicting data. In the latter approach, the last two inferences in Example 5 would not be possible.

For universal integrity constraints, the number of repairs of a finite database is also finite. However, referential integrity constraints may lead to infinitely many repairs, c.f., Example 4. Having infinitely many repairs is a problem for those approaches to computing consistent query answers that construct a representation of all repairs, as do the approaches based on logic programming (Section 4). Therefore, they use a slightly different notion of repair by allowing tuples with *nulls* to be inserted into the database. This reflects common SQL2 database practice. But that approach does not always work, as the *entity integrity* constraint, inherent in the relational data model, prevents null values from appearing in the primary key.

Example 6. Consider Example 4 again. The infinitely many repairs can be replaced by a single repair

Personnel	*SSN*	*Name*
	123456789	*null*
	555555555	*Smith*

Manager	*SSN*
	123456789
	555555555

only if it is the *SSN* attribute which is designated the primary key, not the *Name* attribute (which still remains a key). □

One can also avoid dealing with infinitely many repairs by restricting repairs to be subsets of the original instance, as in [28].

If a different notion of repair than that from Definition 2 is used, the notion of consistent query answer changes too. In general, the more restricted the repairs, the stronger the consistent query answers, as illustrated by the following example.

Example 7. Consider a database schema consisting of two relations $P(AB)$ and $S(C)$. The integrity constraints are: the FD $A \rightarrow B$ and the IND $B \subseteq C$. Assume the database instance $r_1 = \{P(a, b), P(a, c), S(b)\}$. Then there is only one repair $r_2 = \{P(a, b), S(b)\}$ which is a subset of r_1. On the other hand, under Definition 2 there is one more repair $r_3 = \{P(a, c), S(b), S(c)\}$. Therefore, in the first case $P(a, b)$ is consistently true in the original instance r_1, while in the second case it is not. Note that $P(a, c)$ is not consistently true in r_1 either. Therefore, $P(a, b)$ and $P(a, c)$ are treated symmetrically from the point of view of consistent query answering. However, intuitively there is a difference between them. Think of A being the person's name, B her

address and S a list of valid addresses. Then only under the more restricted notion of repair would the single valid address be returned as a consistent answer. □

In the sequel, we will mostly use the notion of repair from Definition 2, clearly indicating the cases where a different notion is applied.

2.4 Computing CQAs

What we have so far is a semantic definition of consistent query answer in a (possibly inconsistent) database, based on the notion of database repair. However, retrieving CQAs via the computation of *all* database repairs is not feasible. Even for FDs the number of repairs may be too large.

Example 8. Consider the functional dependency $A \rightarrow B$ and the following family of relation instances r_n, $n > 0$, each of which has $2n$ tuples (represented as columns) and 2^n repairs:

r_n							
A	a_1	a_1	a_2	a_2	$\cdots$	a_n	a_n
B	b_0	b_1	b_0	b_1	$\cdots$	b_0	b_1

□

Therefore, we develop various methods for computing CQAs *without explicitly computing all repairs*. Such methods can be split in two categories:

1. *Query transformation.* Given a query Q and a set of integrity constraints IC, construct a query Q' such that for every database instance r the set of answers to Q' in r is equal to the set of consistent answers to Q in r w.r.t. IC. This approach was first proposed in [3] for first-order queries. In that case the transformed query is also first-order, thus after a straightforward translation to SQL2 it can be evaluated by any relational database engine. Note that the construction of all repairs is entirely avoided. In [24], the implementation of an extended version of the method of [3] was described.
2. *Compact representation of repairs.* Given a set of integrity constraints IC and a database instance r, construct a space-efficient representation of all repairs of r w.r.t. IC, an then use this representation to answer queries. Different representations have been considered in this context:

2.1. Repairs are answer sets of a logic program [4,6,15]. The compact representation is the program, and to obtain consistent answers, one runs the program.

2.2. Repairs are some distinguished minimal models of a theory written in annotated predicate logic [8,14].

2.3. Repairs are maximal independent sets in a hypergraph whose nodes are database tuples and edges consist of sets of tuples participating in a violation of a denial constraint. This approach has been applied in [28] to quantifier-free first-order queries and in [5,7] to aggregation queries.
2.4. The interaction of the database instance and the integrity constraints is represented as an analytical tableau that becomes closed due to the mutual inconsistency of the database and the integrity constraints. The implicit "openings" of the tableau are the repairs [19]. Implementation issues around consistent query answering based on analytic tableaux for non-monotonic reasoning are discussed in [20].

In the next sections we describe some of these approaches.

3 Query Transformation

Here we consider first-order queries and universal integrity constraints. Given a query, we rewrite it, preserving the original database instance. The query is transformed by qualifying it with appropriate information derived from the interaction between the query and the integrity constraints. This forces the satisfaction of the integrity constraints and makes it possible to discriminate between the tuples in the answer set. The technique is inspired by *semantic query optimization* [25].

More precisely, given a query $\varphi(\bar{x})$, a new query $T^{\omega}(\varphi(\bar{x}))$ is computed by iterating an operator T which transforms a query by conjoining to each database literal appearing in it the corresponding *residue*, until a fixed point is reached. (If there is no residue, then $T(Q) = Q$.) The residue of a database literal forces the satisfaction of the integrity constraints for the tuples satisfying the literal. The residues of a literal are obtained by resolving the literal with the integrity constraints.

Example 9. Consider the following integrity constraints:

$$IC \ = \{\forall x.[R(x) \vee \neg P(x) \vee \neg Q(x)],\ \forall x.[P(x) \vee \neg Q(x)]\}$$

and the query $Q(x)$. The residue of $Q(x)$ wrt the first constraint is $(R(x) \vee \neg P(x))$, because if $Q(x)$ is to be satisfied, then that residue has to be true if the constraint is to be satisfied too. Similarly, the residue of $Q(x)$ wrt the second constraint is $P(x)$. In consequence, instead of the query $Q(x)$, one rather asks the transformed query $Q(x) \wedge (R(x) \vee \neg P(x)) \wedge P(x)$. The literal $\neg Q(x)$ does not have any residues wrt the given integrity constraints, because the integrity constraints do not constrain it. □

If we want the CQAs to an $\mathcal{L}$-query $\varphi(\bar{x})$ in r, we rewrite the query into the new $\mathcal{L}$-query $T^{\omega}(\varphi(\bar{x}))$, and we pose $T^{\omega}(\varphi(\bar{x}))$ to r as an ordinary query. We expect that for every ground tuple $\bar{t}$:

$$r \models_{IC} \varphi(\bar{t}) \ \equiv \ r \models T^{\omega}(\varphi(\bar{t})).$$

We explain later under what conditions this equivalence holds.

Example 10. (Example 1 continued) The FD f_1 can be written as the $\mathcal{L}$-formula

$$f_1: \ \forall xyz.\, [\neg \mathit{Employee}(x,y) \ \vee \ \neg \mathit{Employee}(x,z) \ \vee \ y = z]. \tag{2}$$

If we are given the query $Q(x,y): \ \mathit{Employee}(x,y)$, we expect to obtain the consistent answers: $(V.\mathit{Smith},\ 3000)$, $(M.\mathit{Stowe},\ 7000)$, but not $(J.\mathit{Page},\ 5000)$ or $(J.\mathit{Page},\ 8000)$.

The residue obtained by resolving the query with the FD f_1 is

$$\forall z.\, [\neg \mathit{Employee}(x,z) \ \vee \ y = z].$$

Note that we get the same residue by resolving the query with the first or the second literal of the constraint. Thus, the rewritten query $T(Q(x,y))$ is as follows:

$$T(Q(x,y)) := \mathit{Employee}(x,y) \ \wedge \ \forall z.\, [\neg \mathit{Employee}(x,z) \ \vee \ y = z],$$

and returns exactly $(V.\mathit{Smith},\ 3000)$ and $(M.\mathit{Stowe},\ 7000)$ as answers, i.e., the consistent answers to the original query. □

In general, T needs to be iterated, because we may need to consider the residues of residues and so on. In consequence, depending on the integrity constraints and the original query, we may need to iterate T until the infinite fixed point T^ω is obtained. In Example 10, this was not necessary, because the literal $\neg \mathit{Employee}(x,z)$ in the appended residue does not have a residue wrt f_1 itself. We stop after the first iteration.

Example 11. (Example 9 continued) The following are the sets of residues for the relevant literals (the other literals have no residues):

Literal	*Residue*
$P(x)$	$: \{R(x) \vee \neg Q(x)\}$
$Q(x)$	$: \{R(x) \vee \neg P(x), \quad P(x)\}$
$\neg P(x)$	$: \{\neg Q(x)\}$
$\neg R(x)$	$: \{\neg P(x) \vee \neg Q(x)\}$.

The query is transformed into $T(Q(x)) = Q(x) \wedge (R(x) \vee \neg P(x)) \wedge P(x)$. Now, we apply T again, to the appended residues, obtaining

$$\begin{aligned} T^2(Q(x)) &= Q(x) \wedge (T(R(x)) \vee T(\neg P(x))) \wedge T(P(x)) \\ &= Q(x) \wedge (R(x) \vee (\neg P(x) \wedge \neg Q(x))) \wedge P(x) \wedge (R(x) \vee \neg Q(x)). \end{aligned}$$

And once more

$$T^3(Q(x)) = Q(x) \wedge \ (R(x) \vee (\neg P(x) \wedge T(\neg Q(x)))) \wedge \\ P(x) \wedge (T(R(x)) \vee T(\neg Q(x))).$$

Since $T(\neg Q(x)) = \neg Q(x)$ and $T(R(x)) = R(x)$, we obtain $T^2(Q(x)) = T^3(Q(x))$, and we have reached a fixed point. □

The important properties of the transformation-based approach are: soundness, completeness and termination [3]. *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 $m \geq n$, $\forall \bar{x}(T^n(Q(\bar{x})) \equiv T^m(Q(\bar{x}))$ is a valid formula.

Reference [3] defines some very general sufficient conditions for soundness of the transformation-based approach, encompassing essentially all integrity constraints that occur in practice. Completeness is much harder to achieve. Reference [3] proves completeness of the transformation-based approach for queries that are conjunctions of literals and binary, generic integrity constraints. (A constraint is *generic* if no ground database literal is a logical consequence of it.) For example, we may have the query $R(u, v) \wedge \neg P(u, v)$, and the binary integrity constraints

$$IC = \{\forall x, y.[\neg P(x, y) \vee R(x, y)], \forall x, y, z.[\neg P(x, y) \vee \neg P(x, z) \vee y = z]\}.$$

However, with disjunctive or existential queries we may lose completeness.

Example 12. In Example 10, if we pose the ground disjunctive query

$$Q\colon \mathit{Employee}(J.\mathit{Page},\ 5000) \vee \mathit{Employee}(J.\mathit{Page},\ 8000),$$

the straightforward application of operator T produces the rewritten query $T(Q)$:

$$(\mathit{Employee}(J.\mathit{Page}, 5000)\ \wedge\ \forall z\ (\neg \mathit{Employee}(J.\mathit{Page}, z)\ \vee\ z = 5000)) \vee \\ (\mathit{Employee}(J.\mathit{Page}, 8000)\ \wedge\ \forall z\ (\neg \mathit{Employee}(J.\mathit{Page}, z)\ \vee\ z = 8000)).$$

that has the answer (truth value) *false* in the original database instance, but, according to the definition of consistent answer, is consistently true in this instance. □

Termination can be guaranteed syntactically if there is an n such that $T^n(Q(\bar{x}))$ and $T^{n+1}(Q(\bar{x}))$ are syntactically the same. Reference [3] shows that this property holds for any kind of queries iff the set of integrity constraints IC is *acyclic*, where IC is acyclic if there exists a function

$$f : \{P_1, \ldots, P_n, \neg P_1, \ldots, \neg P_n\} \longrightarrow \mathbb{N},$$

such that for every constraint

$$\forall(\bigvee_{i=1}^{k} l_i(\bar{x}_i) \vee \psi(\bar{x})) \in IC$$

and every $1 \leq i, j \leq k$, if $i \neq j$ then $f(\neg l_i) > f(l_j)$. Here f is the level mapping, similar to the mappings associated with stratified or hierarchical logic programs, except that complementary literals get values independently of each other. Any set of denial constraints – thus also FDs – is acyclic.

For example, termination is syntactically guaranteed for any query if

$$IC = \{\forall x, y.[\neg P(x, y) \vee R(x, y)], \forall x, y, z.[\neg P(x, y) \vee \neg P(x, z) \vee y = z]\}.$$

Reference [3] provides further, non-syntactic sufficient criteria for termination of the transformation-based approach. In particular, termination for multivalued dependencies is obtained.

In [24], an implementation of the operator T^ω is presented. The implementation is done on top of the XSB deductive database system [91], whose tabling techniques make it possible to keep track of previously computed residues and their subsumption. In this way redundant computation of residues is avoided and termination is detected for a wider class of integrity constraints than those presented in [3]. Using XSB allows also a real interaction with the IBM DB2 DBMS.

The query transformation approach to CQAs – as presented in [3,24] – has some limitations. First of all, the methodology is designed to handle only universal integrity constraints, while existential quantifiers are necessary for specifying referential integrity constraints. Furthermore, as we have shown the transformation-based approach fails (it is sound but not complete) for disjunctive or existentially-quantified queries. This failure can be partially explained by complexity-theoretic reasons. Except for very restricted classes of constraints and queries, adding an existential quantifier leads to co-NP-completeness of CQAs. This issue is discussed in more depth in Section 5.

4 Specifying Database Repairs

So far we have presented a model-theoretic definition of CQAs and a computational methodology to obtain such answers for some classes of queries and integrity constraints. Nevertheless, what is still missing is a logical specification $Spec_r$ of *all database repairs* of an instance r, satisfying the following property for all queries Q and tuples $\bar{t}$:

$$Spec_r \vdash Q(\bar{t}) \quad \equiv \quad r \models_{IC} Q(\bar{t}), \tag{3}$$

where $\vdash$ is a new, suitable consequence relation. If we had such a specification, we could consistently answer every query $Q(\bar{x})$ by asking for those $\bar{t}$ such that $Spec_r \vdash Q(\bar{t})$.

As the following example shows, $\vdash$ has to be *non-monotonic*.

Example 13. The database containing the table

Employee	*Name*	*Salary*
	J.Page	5000
	V.Smith	3000
	M.Stowe	7000

is consistent wrt the FD f_1 of Example 1. In consequence, the set of CQAs to the query $Q(x, y) : \ Employee(x, y)$ is

$$\{(J.Page,\ 5000), (V.Smith,\ 3000), (M.Stowe,\ 7000)\}.$$

If we add the tuple $(J.Page,\ 8000)$ to the database, the set of CQAs to the same query is reduced to

$$\{(V.Smith,\ 3000),\ (M.Stowe,\ 7000)\}.$$

□

A specification $Spec_r$ may provide new ways of computing CQAs and shed some light on the computational complexity of this problem.

4.1 Logic programs

We show here how to specify the database repairs of an inconsistent database r by means of a logic program Π_r [4,6]. In order to pose and answer a first-order query $Q(\bar{x})$, a stratified logic program $\Pi(Q)$ plus a new goal query atom $G(\bar{x})$ is obtained by a standard methodology [80,1], and the query $G(\bar{x})$ is evaluated against the program $\Pi_r \cup \Pi(Q)$. The essential part is the program Π_r.

The first observation is that when a database is repaired most of the data *persists*, except for some tuples. More precisely, by default all the positive and implicit negative data persist from r to the repairs, except for some tuples that have to added or removed to restore the consistency of the database. In order to capture this idea, we may use *logic programs with exceptions* [69], in this case containing:

- *default rules* capturing the persistence of the data, and
- *exception rules* stating that certain changes have to be made and the integrity of the database has to be restored.

The exception rules should have higher priority than the default rules. The semantics is that of e-answer sets, based on the answer set semantics for extended disjunctive logic programs [48]. A logic program with exceptions can be eventually translated into an extended disjunctive normal logic program with answer set semantics [69]. Now we give an example of this transformed version, where default rules have been replaced by *persistence rules*, so that

the whole program has an answer set semantics. (In addition to disjunction, the program has two kinds of negation: classical negation $\neg$ and negation-as-failure *not*.)

Example 14. Consider the full inclusion dependency $\forall x.[\neg P(x) \vee Q(x)]$ and the inconsistent database instance $r = \{P(a)\}$. The program Π_r that specifies the repairs of r contains two new predicates, P' and Q', corresponding to the repaired versions of P, Q, resp., and the following sets of rules:

1. *Persistence rules*:

$$P'(x) \leftarrow P(x), not\ \neg P'(x); \qquad Q'(x) \leftarrow Q(x), not\ \neg Q'(x)$$

$$\neg P'(x) \leftarrow not\ P(x), not\ P'(x); \qquad \neg Q'(x) \leftarrow not\ Q(x), not\ Q'(x).$$

   The defaults say that all data persists from the original tables to their repaired versions.
2. *Triggering exception*: $\neg P'(x) \vee Q'(x) \leftarrow P(x),\ not\ Q(x)$.
   This rule is needed as a first step towards the repair of r. It states that in order to "locally" repair the constraint by deleting $P(x)$ or inserting $Q(x)$.
3. *Stabilizing exceptions*: $Q'(x) \leftarrow P'(x)$; $\neg P'(x) \leftarrow \neg Q'(x)$.
   The rules say that eventually the constraint has to be satisfied in the repairs. This kind of exception rules are important if there are interacting integrity constraints and local repairs alone are not enough.
4. *Database facts*: $P(a)$.

If we instantiate the rules in all possible ways in the underlying domain, we obtain a ground program Π_r. A set of ground literals M is an answer set of Π_r if it is a minimal model of Π, where $\Pi = \{A_1 \vee \cdots \vee A_n \leftarrow B_1, \cdots, B_m \mid A_1 \vee \cdots \vee A_n \leftarrow B_1, \cdots, B_m, not\ C_1, \cdots, not\ C_k \in \Pi_r$ and $C_i \notin M$ for $1 \leq i \leq k\}$. If M has complementary literals, then M is a trivial answer set containing all ground literals.

In this example, the answer sets of the program correspond to the expected database repairs: $\{\underline{\neg P'(a)}, \neg Q'(a),\ P(a)\}$; $\{P'(a),\ \underline{Q'(a)}, P(a)\}$. The first one indicates through the underlined literal that $P(a)$ has to be deleted from the database; the second one – that $Q(a)$ has to be inserted in the database. □

In [6], it is proved that for the class of binary integrity constraints (defined in Section 2), there exists a one-to-one correspondence between answer sets and database repairs. In consequence, in (3) we can take $Spec_r$ as a the appropriate extended disjunctive logic program and the notion of logical consequence there as being true wrt all answer sets of the program (i.e. the skeptical answer set semantics).

¿From the correspondence results just mentioned, we can obtain a method to compute database repairs by using any implementation of the answer set semantics for extended disjunctive logic programs. To compute CQAs, one needs to have a way to obtain atoms true in every answer set of the logic program. In [6] the experiments with the deductive database system *DLV* [39] are reported. It is also possible to extend the methodology to include referential integrity constraints containing existentially quantified variables [4,6].

The logic programming approach is very general since 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.

Logic programs for repairing databases and computing CQAs wrt arbitrary universal constraints have been independently introduced in [56]. That work is further discussed in Section 7.

4.2 Annotated logics

As explained at the beginning of this section, we would like to have a logical specification of database repairs. Such a specification must contain information about the database and the integrity constraints – two pieces of information that will be mutually inconsistent if the database does not satisfy the integrity constraints. So including them into a classical first-order theory would lead to an inconsistent theory and the trivialization of reasoning. In consequence, if we want a first-order theory, we have to depart from classical logic, moving to non-classical logic, where reasoning in the presence of classical inconsistencies does not necessarily collapse. Following [8], we show here how to generate a consistent first-order theory with a non-classical semantics. We use *Annotated Predicate Calculus* (APC) [67].

In APC, database *atoms* are annotated with truth values taken from a truth-value lattice. The most common annotations are: true ($\mathbf{t}$), false ($\mathbf{f}$), contradictory ($\top$), and unknown ($\bot$). In [8] a lattice was used to capture the preference for integrity constraints when they conflict with the data: the integrity constraints cannot be given up but the database can be repaired. The new truth values in the lattice are:

- *Database values*: $\mathbf{t_d}$ and $\mathbf{f_d}$, used to annotate the atoms in the original database, resp. outside of it.
- *Constraint values*: $\mathbf{t_c}$ and $\mathbf{f_c}$, used to annotate, depending of their sign, the database literals appearing in the disjunctive normal form of the integrity constraints. The built-in atoms appearing in the integrity constraints are annotated with the classical annotations $\mathbf{t}$ and $\mathbf{f}$.
- *Advisory values*: $\mathbf{t_a}$ and $\mathbf{f_a}$, used to solve the conflicts between the database and the integrity constraints, always in favor of the integrity constraints that are not to be given up, whereas the data is subject to

changes. This is represented in the lattice *Latt* in Figure 1. Intuitively, if a ground atom becomes annotated with both $\mathbf{t_d}$ and $\mathbf{f_c}$, then it gets the value $\mathbf{f_a}$ (the least upper bound of the first two values in the lattice), meaning that the advice is to make it false, as suggested by the integrity constraints. That is, the facts for which the advisory truth values $\mathbf{f_a}$ and $\mathbf{t_a}$ are derived are to be removed from, resp. inserted into, the database in order to satisfy the integrity constraints.

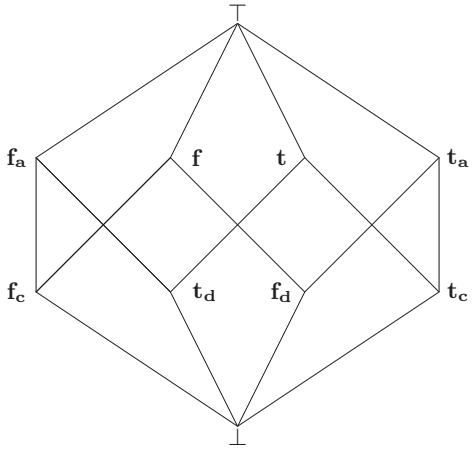

Fig. 1. The truth-value lattice *Latt*

In this lattice, the top element is $\top$, that is reached as the least upper bound (lub) of any pair of contradictory annotations. The annotations $\mathbf{t_d}$ and $\mathbf{f_c}$, for example, are not considered definitely contradictory (i.e. with lub $\top$) if we can still make them compatible by passing to their lub $\mathbf{f_a}$. If there is no conflict between a data and a constraint annotation, then we pass to their lubs, i.e. $\mathbf{t}$ or $\mathbf{f}$.

Now, both the database r and the integrity constraints IC, with the appropriate annotations taken from the lattice, can be embedded into a single and consistent APC theory $Th(r, IC)$. We show this embedding by means of an example.

Example 15. (Example 1 continued) The integrity constraint

$$\forall x, y, z.[\neg \mathit{Employee}(x, y) \ \vee \ \neg \mathit{Employee}(x, z) \ \vee \ y = z]$$

is translated into

$$\forall x, y, z.[\mathit{Employee}(x, y) : \mathbf{f_c} \quad \vee \quad \mathit{Employee}(x, z) : \mathbf{f_c} \quad \vee \quad y = z : \mathbf{t}].$$

Each of the database facts is annotated, e.g. $\mathit{Employee}(J.\mathit{Page}, 5000)$ as $\mathit{Employee}(J.\mathit{Page}, 5000) : \mathbf{t_d}$. We also introduce annotated axioms representing the unique names assumption, and the closed world assumption [87]. In

this way we generate the annotated first-order theory $Th(r, IC)$. □

As mentioned before, navigation in the lattice and an adequate definition of APC formula satisfaction help solve the conflicts between the database and the integrity constraints. The notion of formula satisfaction in a Herbrand interpretation I (now containing annotated ground atoms) is defined as in classical first-order logic, except for atomic formulas. By definition, for such formulas $I \models p{:}\mathbf{s}$, with $\mathbf{s} \in Latt$, iff for some $\mathbf{s}'$ such that $\mathbf{s} \leq_{Latt} \mathbf{s}'$, $p{:}\mathbf{s}' \in I$.

In [8] it is shown that for every database r, there is a one-to-one correspondence between the repairs of r w.r.t. IC and the models of $Th(r, IC)$ that make true a minimal set of atoms annotated with $\mathbf{t_a}$ or $\mathbf{f_a}$ (corresponding to the fact that a minimal set of database atoms is changed). In consequence, the specification $Spec_r$, postulated in (3) at the beginning of this section, is simply $Th(r, IC)$ and the corresponding (non-monotonic) notion of consequence is truth in all $\{\mathbf{t_a}, \mathbf{f_a}\}$-minimal annotated models of the theory. The approach of [8] produces from $Th(r, IC)$ a set of *advisory clauses* that are then processed by specialized algorithms. The approach is applicable to queries that are conjunctions or disjunctions of positive literals, and to universal constraints.

4.3 Logic programs with annotation constants

In [15] a method to obtain a disjunctive logic program $\Pi^{ann}(r, IC)$ from $Th(r, IC)$ is presented. This program, having a stable model semantics, specifies the database repairs. The program has annotations as additional predicate arguments, thus it is a standard, not an annotated [68], logic program, and the standard results and techniques apply to it. We give here an example only.

Example 16. Consider the same database r and integrity constraints IC as in example 14. The logic program should have the effect of repairing the database. Single, local repair steps are obtained as before by deriving the annotations $\mathbf{t_a}$ or $\mathbf{f_a}$. This is done when each constraint is considered in isolation, but there may be interacting integrity constraints, and the repair process may take several steps and should stabilize at some point. In order to achieve this, we need additional, auxiliary annotations $\mathbf{t}^\star$, $\mathbf{f}^\star$, $\mathbf{t}^{\star\star}$, and $\mathbf{f}^{\star\star}$ that are new, special constants in the language.

The annotation $\mathbf{t}^\star$, for example, groups together the annotations $\mathbf{t_d}$ and $\mathbf{t_a}$ for the same atom (the rules 1. and 4. below). This new, derived annotation can be used to provide a feedback to the bodies of the rules that produce the local, single repair steps, so that a propagation of changes is triggered (the rule 2. below). The annotations $\mathbf{t}^{\star\star}$ and $\mathbf{f}^{\star\star}$ are just used to read off the literals that are inside (resp. outside) a repair. This is achieved by means of the rules 6. below, that are used to interpret the models as database repairs. The following is the program $\Pi^{ann}(r, IC)$:

1. $P(x, \mathbf{f}^\star) \leftarrow P(x, \mathbf{f_a}).$ $\quad P(x, \mathbf{t}^\star) \leftarrow P(x, \mathbf{t_a}).$ $\quad P(x, \mathbf{t}^\star) \leftarrow P(x, \mathbf{t_d}).$

   $Q(x, \mathbf{f}^\star) \leftarrow Q(x, \mathbf{f_a}).$ $\quad Q(x, \mathbf{t}^\star) \leftarrow Q(x, \mathbf{t_a}).$ $\quad Q(x, \mathbf{t}^\star) \leftarrow Q(x, \mathbf{t_d}).$

2. $P(x, \mathbf{f_a}) \vee Q(x, \mathbf{t_a}) \leftarrow P(x, \mathbf{t}^\star), Q(x, \mathbf{f}^\star).$
3. $P(a, \mathbf{t_d}) \leftarrow.$
4. $P(x, \mathbf{f}^\star) \leftarrow \mathit{not}\ P(x, \mathbf{t_d}).$ $\quad Q(x, \mathbf{f}^\star) \leftarrow \mathit{not}\ Q(x, \mathbf{t_d}).$
5. $\leftarrow P(\bar{x}, \mathbf{t_a}), P(\bar{x}, \mathbf{f_a}).$ $\quad \leftarrow Q(\bar{x}, \mathbf{t_a}), Q(\bar{x}, \mathbf{f_a}).$
6. $P(x, \mathbf{t}^{\star\star}) \leftarrow P(x, \mathbf{t_a}).$ $\quad P(x, \mathbf{t}^{\star\star}) \leftarrow P(x, \mathbf{t_d}),\ \mathit{not}\ P(x, \mathbf{f_a}).$

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

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

   $Q(x, \mathbf{f}^{\star\star}) \leftarrow Q(x, \mathbf{f_a}).$ $\quad Q(x, \mathbf{f}^{\star\star}) \leftarrow \mathit{not}\ Q(x, \mathbf{t_d}),\ \mathit{not}\ Q(x, \mathbf{t_a}).$

The rule 2. is the only rule dependent on the integrity constraints. It says how to repair the constraint when an inconsistency is detected. If there were other integrity constraints interacting with this constraint, having passed to the annotations $\mathbf{t}^\star$ and $\mathbf{f}^\star$ will allow the system to keep repairing the constraint if it becomes violated due to the repair of a different constraint. Rules in 3. contain the database atoms. Rules 4. capture the closed world assumption. The rules in 5. are denial constraints for coherence. That is, coherent models do not contain atoms annotated with both $\mathbf{t_a}$ and $\mathbf{f_a}$.

Stable models are defined exactly as the answer sets in Example 14, but considering sets of ground atoms only since in the programs with annotations there is no classical negation.

The program in this example has two stable models:

$$\{P(a, \mathbf{t_d}), P(a, \mathbf{t}^\star), Q(a, \mathbf{f}^\star), Q(a, \mathbf{t_a}), \underline{P(a, \mathbf{t}^{\star\star})}, Q(a, \mathbf{t}^\star), \underline{Q(a, \mathbf{t}^{\star\star})}\}$$

and

$$\{P(a, \mathbf{t_d}), P(a, \mathbf{t}^\star), P(a, \mathbf{f}^\star), Q(a, \mathbf{f}^\star), \underline{P(a, \mathbf{f}^{\star\star})}, \underline{Q(a, \mathbf{f}^{\star\star})}, P(a, \mathbf{f_a})\},$$

the first one saying, through its underlined atoms that $Q(a)$ is to be inserted into the database; the second one – that $P(a)$ is to be deleted. □

In [15], a one-to-one correspondence between the stable models of the programs $\Pi^{ann}(r, IC)$ and the repairs of r wrt IC is established. Consistent answers can thus be obtained by "running" a query program together with the repair program $\Pi^{ann}(r, IC)$, under the skeptical stable model semantics.

The programs with annotations obtained are simpler than those in section 4.1 in the sense that they contain one change triggering rule per constraint (rule 4. in the example), whereas the natural extension to arbitrary universal constraints of the approach in Section 4.1 may produce programs with the number of rules which is exponential in the number of disjuncts in the

disjunctive normal forms of the (universal) integrity constraints [6,56]. The method of [15] can also capture repairs of referential integrity constraints (under the notion of repair allowing tuples with nulls, as discussed in Section 2). Thus, the approach in [15] is the most general considered so far, since it applies to arbitrary first-order queries, and arbitrary universal or referential integrity constraints (with the exception of the cases that may lead to the violations of the entity integrity constraint, c.f., Example 6).

In some cases, optimizations of the program are possible. For example, the program we just gave is *head-cycle free* [16]. In consequence, it can be transformed into a non-disjunctive normal program, reducing the complexity of its evaluation from Π_2^p to co-NP [74,34]. Not every repair program with annotations will be head-cycle free though, because there are some limitations imposed by the intrinsic computational complexity of the problem of consistent query answering.

5 Computational Complexity

We summarize here the results about the computational complexity of consistent query answers [3,7,29,28]. We will adopt the *data complexity* assumption [1,65,94] which measures the complexity of the problem as a function of the number of tuples in a given database instance. The given query and integrity constraints are considered fixed.

The query transformation approach [3] – in the cases where it terminates – provides 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 CQAs: 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. For example, in Example 8 the query $R(x, y)$ will be transformed (similarly to the query in Example 10) to another first-order query and evaluated in PTIME, despite the presence of an exponential number of repairs. However, the query transformation approach is sound, complete and terminating only for restricted classes of queries and constraints. More specifically, the results of [3] imply that for binary denial constraints and full inclusion dependencies consistent answers can be computed in PTIME for queries that are conjunctions of literals. The logic programming approaches described in Section 4 do not have good asymptotic complexity properties, since they are all based on Π_2^p-complete classes of logic programs [34]. So it was an open question how far the boundary between tractable and intractable can be pushed in this context.

The paper [28] ([29] is an earlier version containing only some of the results) shows how the complexity of computing CQAs depends on the *type* of the constraints considered, their *number*, and the *size* of the query. Several new classes for which consistent query answers are in PTIME are identified:

- ground quantifier-free queries and arbitrary denial constraints;
- closed simple (without repeated relation symbols) conjunctive queries, and functional dependencies, with at most one FD per relation;
- ground quantifier-free or closed simple conjunctive queries, and key functional dependencies and foreign key constraints, with at most one key per relation.

Additionally, the paper [28] analyzes the data complexity of *repair checking*: the problem of testing whether one database is a repair of another. (The paper [28] makes the assumption that repairs are subsets of the original instance.) It is shown that repair checking is in PTIME for all the above classes of constraints, as well as for arbitrary FDs together with acyclic INDs. The results obtained are tight in the sense that relaxing any of the above restrictions leads to co-NP-hard problems. (This, of course, does not preclude the possibility that introducing *additional*, orthogonal restrictions could lead to more PTIME cases.) To complete the picture, it is shown that for arbitrary sets of FDs and INDs repair checking is co-NP-complete and consistent query answers is Π_2^p-complete.

We outline now the proof of the first result listed above, since it is done using a technique different from query transformation. We introduce first the notion of a *conflict hypergraph* that will serve as a succinct representation of all the repairs of a given instance.

Definition 5. The *conflict hypergraph* $\mathcal{G}_{F,r}$ is a hypergraph whose set of vertices is the set $\Sigma(r)$ of facts of an instance r and whose set of edges consists of all the sets

$$\{P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots P_l(\bar{t}_l)\}$$

such that $P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots P_l(\bar{t}_l) \in \Sigma(r)$, and there is a constraint

$$\forall \bar{x}_1, \bar{x}_2, \ldots \bar{x}_l.\ [\neg P_1(\bar{x}_1)\ \vee\ \neg P_2(\bar{x}_2)\ \vee\ \ldots\ \vee\ \neg P_l(\bar{x}_l)\ \vee\ \phi(\bar{x}_1, \bar{x}_2, \ldots \bar{x}_l)]$$

in F such that $P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots P_l(\bar{t}_l)$ violate together this constraint, which means that there exists a substitution ρ such that $\rho(\bar{x}_1) = \bar{t}_1, \rho(\bar{x}_2) = \bar{t}_2, \ldots \rho(\bar{x}_l) = \bar{t}_l$ and that $\phi(\bar{t}_1, \bar{t}_2, \ldots \bar{t}_l)$ is false.

By an *independent set* in a hypergraph we mean a subset of its set of vertices which does not contain any edge. Clearly, each repair of r w.r.t. F corresponds to a maximal independent set in $\mathcal{G}_{F,r}$.

We prove here that for every set F of denial constraints and ground quantifier-free query Φ, the data complexity of checking whether Φ is consistently true w.r.t. F in an instance r is in PTIME. We assume the sentence is in CNF, i.e., of the form $\Phi = \Phi_1 \wedge \Phi_2 \wedge \ldots \Phi_l$, where each Φ_i is a disjunction of ground literals. Φ is true in every repair of r if and only if each of the clauses Φ_i is true in every repair. So it is enough to provide a polynomial algorithm which will check if a given ground clause is consistently true.

It is easier to think that we are checking if for a ground clause *true* is **not** a consistent answer. This means that we are checking, whether there exists a repair r' in which $\neg\Phi_i$ is true for some i. But $\neg\Phi_i$ is of the form $P_1(\bar{t}_1) \wedge P_2(\bar{t}_2) \wedge \ldots \wedge P_m(\bar{t}_m) \wedge \neg P_{m+1}(\bar{t}_{m+1}) \wedge \ldots \wedge \neg P_n(\bar{t}_n)$, where the $\bar{t}_j$'s are tuples of constants. WLOG, we assume that all the facts in the set $\{P_1(\bar{t}_1), \ldots, P_n(\bar{t}_n))\}$ are mutually distinct.

The nonderministic algorithm selects for every j, $m+1 \leq j \leq n$, $\bar{t}_j \in r$, an edge $E_j \in \mathcal{G}_{F,r}$ such that $\bar{t}_j \in E_j$. Additionally the following global condition needs to be satisfied: *there is no edge* $E \in \mathcal{G}_{F,r}$ *such that* $E \subseteq r'$ *where*

$$r' = \{\bar{t}_1, \ldots, \bar{t}_m\} \cup \bigcup_{m+1 \leq j \leq n, \bar{t}_j \in r} (E_j - \{\bar{t}_j\}).$$

If the selection succeeds, then a repair in which $\neg\Phi_i$ is true can be built by adding to r' new tuples from r until the set is maximal independent. The algorithm needs $n - m$ 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 PTIME deterministic algorithm.

6 Aggregation Queries

So far we have considered only first-order queries but in databases aggregation queries are also important. In fact, aggregation is essential in scenarios, like data warehousing, where inconsistencies are likely to occur, and keeping inconsistent data may be useful. Only some aggregation queries, e.g. computing a maximum or minimum value of an attribute in a relation can be expressed as first-order queries. Even in this case, due to its syntax, the resulting first-order query cannot be handled by the query transformation methodology described earlier.

We will consider here a restricted scenario: the integrity constraints will be limited to functional dependencies, and the aggregation queries will consist of single applications of one of the standard SQL-2 aggregation operators (`MIN`, `MAX`, `COUNT(*)`, `COUNT(A)`, `SUM`, and `AVG`). Even in this case, it was shown [5] that computing consistent query answers to aggregation queries is a challenging problem.

Example 17. Consider again the instance r of *Employee* from Example 1. It is inconsistent w.r.t. the FD f_1: *Name* $\rightarrow$ *Salary*.

Employee	*Name*	*Salary*
	J.Page	5000
	J.Page	8000
	V.Smith	3000
	M.Stowe	7000

The repairs are:

Employee1	*Name*	*Salary*
	J.Page	5000
	V.Smith	3000
	M.Stowe	7000

Employee2	*Name*	*Salary*
	J.Page	8000
	V.Smith	3000
	M.Stowe	7000

If we pose the query

```
SELECT MIN(Salary) FROM Employee
```

we should get 3000 as a consistent answer: `MIN(Salary)` returns 3000 in each repair. Nevertheless, if we ask

```
SELECT MAX(Salary) FROM Employee
```

then the maximum, 8000, comes from a tuple that participates in the violation of *FD*. Actually, `MAX(Salary)` returns a different value in each repair: 7000 or 8000. Thus, there is no consistent answer in the sense of definition 4. □

We give a new, slightly weakened definition of consistent answer to an aggregation query that addresses the above difficulty.

Definition 6. [5] (a) A consistent answer to an aggregation query Q with respect to a database instance r and a set of integrity constraints F is the minimal interval $I = [a, b]$ such that for every repair r' of r w.r.t. F, the scalar value $Q(r')$ of query Q in r' belongs to I.
(b) The left and right endpoints of the interval I are the *greatest lower bound* (glb) and *least upper bound* (lub), resp., answers to Q in r. □

According to this definition, in Example 17 the interval $[7000, 8000]$ is the consistent answer to the query

```
SELECT MAX(Salary) FROM Employee
```

and 7000 and 8000 are the *glb-answer* and *lub-answer*, resp. Notice that 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 r wrt a set of FDs. The representation of the interval is always polynomially sized, since the numeric values of the endpoints can be represented in binary.

Example 18. Along the lines of Example 8, consider the functional dependency $A \to B$ and the following family of relation instances S_n, $n > 0$:

r_n										
A	a_1	a_1	a_2	a_2	a_3	a_3	$\cdots$	a_n	a_n
B	0	1	0	2	0	4	$\cdots$	0	2^n

The aggregation query `SUM(B)` takes all the exponentially many values between 0 and $2^{n+1} - 1$ in the (exponentially many) repairs of the database [5]. An explicit representation of the possible values the aggregation function would then be exponentially large. Moreover, it would violate the 1NF assumption. On the other hand, the interval representation has polynomial size. □

Next, we consider the complexity of the problem of computing the *glb-* and *lub-answers*. The complexity results are given in terms of data complexity [1,65,94]. For classifying the problems of consistent answering to different aggregation queries in terms of complexity and for finding polynomial time algorithms in tractable cases, it is useful to use a graph representation of the set of all repairs. Because we are dealing with functional dependencies, we can specialize the notion of *conflict hypergraph* (Definition 5) to that of a *conflict graph* (edges contain two vertices).

Example 19. Consider the schema $R(AB)$, and a set F of two functional dependencies $A \rightarrow B$ and $B \rightarrow A$, and the inconsistent instance $r = \{(a_1, b_1), (a_1, b_2), (a_2, b_2), (a_2, b_1)\}$ over this schema. The following is the conflict graph $\mathcal{G}_{F,r}$:

(a_1, b_1) (a_1, b_2)

(a_2, b_1) (a_2, b_2)

The two maximal independent sets $\{(a_1, b_1), (a_2, b_2)\}$ and $\{(a_1, b_2), (a_2, b_1)\}$ correspond to the two possible repairs of the database. □

The paper [7] contains a complete classification of the tractable/intractable cases of the problem of computing consistent query answers (in the sense of Definition 6) to aggregation queries. Its results can be summarized as follows:

- for all the aggregate operators except `COUNT(A)`, the problem is in PTIME if the set of integrity constraints contains at most one non-trivial FD;
- for `COUNT(A)` the problem is NP-complete even for one non-trivial FD (one can encode the HITTING SET problem [47]);
- for more than one non-trivial FD, even the problem of checking whether the glb-answer to a query is $\leq k$ (resp. the problem of checking whether the lub-answer to a query is $\geq k$) is NP-complete.

For the aggregate operators `MIN`, `MAX`, `COUNT(*)` and `SUM` and a single FD, the glb- and lub-answers are computed by SQL2 queries (so this is in a sense an analogue of the query transformation approach for first-order queries discussed earlier). For `AVG`, however, the PTIME algorithm is iterative and cannot be formulated in SQL2.

Example 20. Continuing Example 17, the greatest lower bound answer to the query

```
SELECT MAX(Salary) FROM Employee
```

is computed by the following SQL2 query

```
SELECT MAX(C) FROM
  (SELECT MIN(Salary) AS C
   FROM Employee
   GROUP BY Name).
```

□

In [5,7], some special properties of conflict graphs in restricted cases were identified, paving the way to more tractable cases. For example, for two FDs and the relation schema in Boyce-Codd Normal Form, the conflict graphs are claw-free and perfect [22], and computing lub-answers to `COUNT(*)` queries can be done in PTIME.

Given the intractability results, it seems appropriate to find approximations to consistent answers to aggregation queries. Unfortunately, "maximal independent set" seems to have bad approximation properties [60].

7 Related work

We discuss here related work on dealing with inconsistency in artificial intelligence, databases and logic programming. We will attempt to characterize various approaches along several common dimensions, including:

- *semantics:* What is the underlying notion of inconsistency? Are the notions of repair and consistent query answer supported in any sense?
- *scope:* What classes of databases, integrity constraints and queries can be handled?
- *computational mechanisms:* How is consistent information obtained in the presence of inconsistency?
- *computational complexity.*

To be able to delineate the scope of different approaches, one has to observe whether they are first-order or propositional, and if they are first-order – whether they can be *reduced* to propositional. The approaches presented in this paper so far are first-order. However, they can be reduced to the propositional case for universal integrity constraints and ground queries, since the constraints themselves can be grounded using the constants in the database and the query. For more general classes of queries and constraints, e.g., referential integrity constraints, such a reduction does not apply. Moreover, the

approaches which do not require grounding, e.g., query transformation in Section 3, are preferable from the efficiency point of view.

A specific dimension of the computational mechanisms under consideration is the support for *locality* of inconsistency. Locality in our context means that the consistency violations that are irrelevant to a given query are ignored in the process of obtaining consistent answers to the query. Clearly, locality is desirable but it is supported only by a few approaches. The query transformation approach (Section 3) supports locality, since the violations occurring in the relations not mentioned in the transformed query are irrelevant for the evaluation of this query and are ignored. The approaches based on some form of specification of all repairs (Section 4) do not support locality, because they require the resolution of all violations through the construction of all answer sets (or minimal models). The algorithm described in Section 5 is based on constructing the conflict hypergraph of the given instance and while it does not resolve all the conflicts, it has to detect all of them. Other PTIME algorithms mentioned in that section support locality. It seems possible to refine the non-local approaches mentioned above in such a way as to obtain locality.

7.1 Belief revision and update

Semantically, our approach to consistency handling corresponds to some of the approaches followed by the belief revision/update community [45,46]. Database repairs (Definition 2) coincide with revised models defined by Winslett [96]. Both use the same notion of minimality. Comparing our framework with that of belief revision, we have an empty domain theory, one model: a database instance, and a revision by a set of integrity constraints. The revision of a database instance by the integrity constraints produces new database instances – the repairs of the original database. The scenario adopted by most belief revision papers is thus more general than ours, since such papers typically assume that it is a formula (or, equivalently the set of its models) that is undergoing the revision, and that the domain theory is nonempty. On the other hand, the research on belief revision is typically limited to the propositional case.

Our implicit notion of revision satisfies the postulates (R1) – (R5),(R7) and (R8) introduced by Katsuno and Mendelzon [66]. Dalal [32] postulated a different notion of revision, based on minimizing the cardinality of the set of changes, as opposed to minimizing the set of changes under set inclusion [3,96]. In [6] it is shown how to capture repairs under Dalal's notion of revision by means of logic programs for consistent query answering.

The belief revision community has adopted a notion of inference called *counterfactual inference* [45] that corresponds to our notion of a formula being consistently true. Counterfactual inference is based on the *Ramsey test* for conditionals: a formula $\beta > \gamma$ is a counterfactual consequence of a set of beliefs K if for every closest context in which K is revised in such a way that

β is true, γ is also true. In our case, K is a database, β is the set of integrity constraints, and γ is the query.

Winslett's approach [96] is mainly propositional, but a preliminary extension to the first-order ground case can be found in [30]. Those papers concentrate on the computation of the models of the revised theory, i.e., the repairs in our case. Inference or query answering is not addressed. The complexity of belief revision and counterfactual inference was exhaustively classified by Eiter and Gottlob [40]. The paper [40] deals with the propositional case only. We have outlined above how to reduce – in some cases – consistent query answering to the propositional case by grounding. However, grounding of integrity constraints results in an update formula which is *unbounded*, i.e., whose size depends on the size of the database. This prevents the transfer of any of the PTIME upper bounds from [40] into our framework. Similarly, the lower bounds from [40] require different kinds of formulas from those that we use. The classic paper on updating logical theories by Fagin et al. [41] focuses on the semantics of updates but does not address the computational issues. Moreover, the proposed framework is also limited to the propositional case. It is interesting that [41] proposes yet another notion of *repair minimality* by giving priority to minimizing deletions over minimizing insertions.

The approaches pursued by the belief revision community are non-local in the sense of having to resolve all the inconsistencies in the database, even those that are irrelevant to the query.

7.2 Reasoning in the presence of inconsistency

There are many approaches to inconsistency handling in the literature[1]. Many of them have been proposed by the logic community, the most prominent being the family of *paraconsistent logics* [31,61]. Such logics protect reasoning in the presence of classical inconsistencies from *triviality* – the property that an inconsistent theory entails every formula. Their applicability in the context of inconsistent databases is, however, limited. First, they typically do not address the issue of the special role of integrity constraints whose truth cannot be given up during the inference process. Second, most paraconsistent logic are monotonic, and thus fail to capture the nonmonotonicity inherent in the notion of consistent query answer (Example 13). Third, they are mostly non-local. Below we discuss those paraconsistency-based approaches that are the closest to ours.

Bry [23] was, to our knowledge, the first author to consider the notion of consistent query answer in inconsistent databases. He defined consistent query answers using provability in minimal logic. The proposed inference method is nonmonotonic but fails to capture minimal change (thus Bry's notion of consistent query answer is weaker than ours). Moreover, Bry's approach is entirely proof-theoretic and does not provide a computational mech-

[1] For recent collections of papers see [17,36].

anism to obtain consistent answers to first-order queries. Other formalisms, e.g., [78], are also limited to propositional inference. Moreover, they do not distinguish between integrity constraints and database facts. Thus, if the data in the database violates an integrity constraint, the constraint itself can no longer be inferred (which is not acceptable in the database context).

Example 21. Assume the integrity constraint is $(\neg p \vee \neg q)$ and the database contains the facts p and q. In the approach of Lin [78], $p \vee q$ can be inferred (minimal change is captured correctly) but p, q and $(\neg p \vee \neg q)$ can no longer be inferred (they are all involved in an inconsistency). □

Several papers by Lozinskii, Kifer, Arieli and Avron [9,67,81] studied the problem of making inferences from a possibly inconsistent, propositional or first-order, knowledge base. The basic idea is to infer the classical consequences of all maximal consistent subsets of the knowledge base [81] or all *most consistent* models of the knowledge base [9,67] (where the order on models is defined on the basis of atom annotations drawing values from a lattice or a bi-lattice). This provides a non-monotonic consequence relation but the special role of the integrity constraints (whose truth cannot be given up) is not captured. Also, no computational mechanisms for answering first-order (or aggregation) queries are proposed, neither are computational complexity issues addressed. In section 4, we described how the approach of Kifer and Lozinskii [67] can be adapted to the task of computing consistent query answers.

In [35] a logical framework based on a three-valued logic is used to distinguish between consistent and inconsistent (controversial) information. A database instance is a finite set of tuples, each tuple associated with the value 1 (safe), 0 (false, does not need to be stored) or $\frac{1}{2}$ (controversial). Integrity constraints are expressed in a first-order language and have three-valued semantics. A repair J of I is an instance satisfying a set of integrity constraints IC, which is $\leq_I$-minimal among all the instances satisfying IC, where $\leq_I$ is defined as follows. The distance between I and J is the sum over all tuples u of $|I(u) - J(u)|$, where $I(u)$ and $J(u)$ are the values associated to u in I and J, respectively. Then, $J \leq_I K$ if the distance between I and J is less than or equal to the distance between I and K. Furthermore, in [35] an algorithm for computing repairs is introduced. This algorithm is based on the tableau proof system for the three-valued logic used in the framework. A related approach of Arieli et al. [10] introduces executable specifications of repairs using abductive logic programming [64]. In both approaches, however, no notion analogous to consistent query answers is proposed and no complexity analysis provided.

Pradhan [85,86] introduced a logic for reasoning in the presence of *contestations* that are conflicts of different kinds: logical, semantical, domain dependent, etc. They are declared together with the domain specification which is, e.g., a logical theory or a normal logic program. The logic has a

non-classical, four-valued semantics that allows inferring conflict-free sets of consequences. For example, if conflicts have been declared as classical logical conflicts, no logical contradiction will be found in the set of consequences. A deductive evaluation mechanism is developed for ground queries that are strong consequences of the specification, i.e. that hold in all conflict-free models. Furthermore, it is also shown how to represent – as a set of conflicts – the inconsistency of a deductive database wrt a set of integrity constraints. An interesting approach to integrity constraints in database is taken: they should restrict the possible answers one can get from the database, rather that capture the semantics of the domain or restrict the states of the database. This view is quite compatible with the approach in [3], and could be used as another motivation for it. However, the general deductive system for strong consequences is not explicitly applied nor specialized to consistent (conflict-free) query answering in databases. Complexity issues are not addressed.

Further related treatments of inconsistency have been developed in the areas of knowledge representation [43], and formal specifications in software engineering [12,84].

7.3 Databases

The approaches discussed here and in the next subsection are applicable to relational databases, and first-order queries and integrity constraints.

Asirelli et al. [11] treat integrity constraints as views over a deductive database. In that way, queries can be answered "through the views", in such a way that the resulting answers satisfy the integrity constraints, and answers that do not satisfy them are filtered out. This approach is the closest to the transformation-based approach presented in Section 3 and also supports locality. However, the approach [11] is a deductive, resolution-based, direct query answering method, similar to the approaches to query answering in deductive databases in the presence of integrity constraints [70,90]. Moreover, queries are restricted to be conjunctions of literals. No computational complexity issues are addressed.

Wijsen [95] studies the problem of consistent query answering in the context of universal constraints. In contrast to Definition 2, he considers repairs obtained by modifying individual tuple components. Notice that a modification of a tuple component cannot be necessarily simulated as a deletion followed by an insertion, because this might not be minimal under set inclusion. Wijsen proposes to represent all the repairs of an instance using a single *trustable tableau*. From this tableau, answers to conjunctive queries can be efficiently obtained. It is not clear, however, what is the computational complexity of constructing the tableau, or even whether the tableau is always of polynomial size.

Franconi et al. [42] also discuss repairs based on updating individual values, in the context of a data cleaning application. The aim is to compute all

possible repairs, in this case of a particular kind of databases storing census data, rather that consistent query answering. The issues addressed consist of detecting and solving conflicts inside the database, and conflicts between answers to questionnaires and the intended, declarative semantics of the latter, as opposed to conflicts between data and integrity constraints. This work is a specific case of *data cleaning* [44].

It has been widely recognized that in database integration the integrated data may be inconsistent with the integrity constraints. A typical (theoretical) solution to the problem of database inconsistency in this context is to augment the data model to represent disjunctive information. Different disjuncts correspond to different ways of resolving an inconsistency. The following example explains the need for a solution of this kind.

Example 22. Consider the functional dependency *"every person has a single salary"* in Example 1. It is violated by the first two tuples. Each of those tuples may be coming from a different data source that satisfies the dependency. Thus, both tuples are replaced by their disjunction

$$\mathit{Employee}(J.\mathit{Page}, 5000) \vee \mathit{Employee}(J.\mathit{Page}, 8000)$$

in the integrated database. Now the functional dependency is no longer violated. □

To solve this kind of problem, Agarwal et al. [2] introduced the notion of *flexible relation*, a non-1NF relation that contains tuples with sets of non-key values (with such a set standing for *one* of its elements). This approach is limited to primary key functional dependencies and was subsequently generalized to other key functional dependencies by Dung [37]. In the same context, Baral et al. [13,53] proposed to use disjunctive Datalog, and Lin and Mendelzon [79] tables with OR-objects [62,63]. Agarwal et al. [2] introduced flexible relational algebra to query flexible relations, and Dung [37] introduced flexible relational calculus (a proper subset of the calculus can be translated to flexible relational algebra). The remaining papers did not discuss query language issues, relying on the existing approaches to query disjunctive Datalog or tables with OR-objects. There are several important differences between the above approaches and ours. First, they rely on the construction of a single (disjunctive) instance and the deletion of conflicting tuples. The integrity constraints are used solely for conflict resolution. However, all the conflicts need to be resolved and thus the above approaches are non-local. In our approach, the underlying databases are incorporated into the integrated one *in toto*, without any changes. There is no need for introducing disjunctive information. The integrity constraints are used only during querying. Second, the above approaches do not generalize to arbitrary functional dependencies and other kinds of integrity constraints. Imielinski et al. [63] provide a comprehensive characterization of the computational complexity of evaluating

conjunctive queries in databases with OR-objects. Those results carry over into our framework only in very limited cases, as discussed in [7,28].

Motro [83] addressed the issue of integrating data from possibly mutually inconsistent sources in a fashion different from the above and closer to our approach. He proposed, among others, the notion of *sound* query answers – the answers present in the query result in every source. For functional dependencies and single-literal queries, every sound answer (in Motro's sense) is a consistent answer (in our sense). However, the converse is not true, since a tuple that appears only in a single source will not be a sound answer, while it is a consistent answer if it does not conflict with any other tuple. Also, for general denial constraints, there may be sound answers that are not consistent. The computational mechanism proposed in [83] consists of simply taking the intersection of the query answers in individual sources and thus it is local. No complexity analysis is provided.

Gertz [49,50] described techniques based on model-based diagnosis for detecting causes of inconsistencies in databases and computing the corresponding repairs. However, he didn't address the issue of query answering in the presence of an inconsistency.

Cholvy [27] introduced a deductive approach based on modal logic that allows limiting the impact of inconsistent information that is related to a query. The logic tells apart *sure* and *doubtful* information. From the original inconsistent deductive database that includes integrity constraints, a new database consisting of modal formulas is constructed. There are modalities S and D for the *sure* and *doubtful* formulas. Then, the idea is to derive the sure answers from the deductive system, so query processing consists of constructing a proof in an appropriate modal logic. Integrity constraints are considered at the same level of reliability than data, and in consequence, they could be considered "doubtful". No complexity analysis is provided.

7.4 Logic programming

Greco et al. [56,58] independently developed a logic-programming-based approach to inconsistency handling in databases, alternative to those presented in sections 4.1 and 4.2. In that approach, disjunctive logic programs with stable model semantics are used to specify the sets of changes that lead to database repairs in the sense of [3]. The authors present a general solution based on a compact schema for generating repair programs for universal integrity constraints. The application of such a schema leads to rules whose heads involve essentially all possible disjunctions of database literals that occur together in a constraint. Thus a single constraint can produce exponentially many clauses. The approach of [4,6] can be generalized to non-binary constraints along the same lines. (In contrast to [4,6,56,58], the approach of [15] does not lead to an exponential blowup.) The approach of [56] is concentrated mainly on producing the sets of changes, rather than the repaired

databases explicitly. In particular, there are no persistence rules in the generated program. In consequence, the program cannot be directly used to obtain consistent query answers. An additional contribution of [56] is the notion of *repair constraints* that specify preferences for certain kinds of repairs (e.g., deletion over insertion).

Another approach to database repairs based on logic programming semantics consists of *revision programs* proposed by Marek and Truszczynski [82]. The rules in those programs explicitly declare how to enforce the satisfaction of an integrity constraint, rather than explicitly stating the integrity constraints, e.g.

$$in(a) \leftarrow in(a_1), \ldots, in(a_k), out(b_1), \ldots, out(b_m)$$

has the intended procedural meaning of inserting the database atom a whenever $a_1, \ldots, a_k$ but not $b_1, \ldots, b_m$ are in the database. Also a declarative, stable model semantics is given to revision programs (thus providing also a computational mechanism). Preferences for certain kinds of repair actions can be captured by including the corresponding rules in the revision program and omitting the rules that could lead to other forms of repairs. No notion analogous to consistent query answers is proposed.

There are several proposals for language constructs specifying nondeterministic queries that are related to our approach (*witness* [1], *choice* [51,52,57]). Essentially, the idea is to construct a maximal subset of a given relation that satisfies a given set of functional dependencies. Since there is usually more than one such subset, the approach yields nondeterministic queries in a natural way. Clearly, maximal consistent subsets (choice models [51]) correspond to repairs of Definition 2. Stratified Datalog with choice [51] combines enforcing functional dependencies with inference using stratified Datalog programs. Answering queries in all choice models ($\forall G$-queries [57]) corresponds to our notion of computation of consistent query answers for first-order queries (Definition 4). However, in [57] the former problem is shown to be co-NP-complete and no tractable cases are identified. One of the sources of complexity in this case is the presence of intensional relations defined by Datalog rules. Such relations are absent from our approach. Moreover, the procedure proposed in [57] runs in exponential time if there are exponentially many repairs, as in Example 8. Also, only conjunctions of literals are considered as queries in [57]. Arbitrary first-order or aggregation queries are not studied. Neither is the approach generalized beyond functional dependencies.

Blair and Subrahmanian [21] introduced paraconsistent logic programming. Paraconsistent logic programs have non-classical semantics, inspired by paraconsistent first-order semantics. Kifer and Subrahmanian [68] discussed annotated logic programs with lattice-based, non-classical semantics. Atoms in clauses have annotations, as in [67], but now they may also contain variables and functions, providing a stronger representation formalism. The implementation of annotated logic programs and query answering mechanisms are discussed in [71]. Subrahmanian [92] further generalized annotated

programs, in order to use them for amalgamating databases by resolving potential conflicts between integrated data. For this purpose the product of the lattices underlying each database is constructed as the semantic basis for the integrated database. Conflict resolutions and preferences are captured by means of function-based annotations. Other approaches to paraconsistent logic programming are discussed in [33]. These works do not define notions analogous to repairs and consistent query answers.

8 Open Problems and Ongoing Work

8.1 Flexible repairs

The existing notions of consistent answer as presented in [3–5,56,58,72] are based on the notion of database repair from [3]. Database repairs of an inconsistent database instance are new instances that satisfy the integrity constraints, but differ from the original instance by a minimal set of *whole* database tuples, where minimality is understood under set inclusion. In Section 2 we mentioned some alternative notions of repair. In particular, in some situations it may be more natural to consider more *flexible* repairs that allow modifications of individual tuple components [95].

Other alternatives, independently on how repairs are defined, should consider more flexibility wrt the class of all repairs. For example, considering an answer as consistent if it is true in the majority of the database repairs, or true in some preferred repairs, under some predefined notion of preference. Majority-based approaches to consistency have been studied in [79] and [81] in the context of data integration. The whole issue of preferences for certain changes and repairs remains still to be investigated. Some work is this direction is presented in [56].

8.2 Data integration

Assume we have a collection of (materialized) data sources $S_1, \ldots, S_n$, and a global, virtual database G, that integrates data from $S_1, \ldots, S_n$. According to the *local-as-view* approach [75,83,93], we can look at the data sources, S_i, as *views* of the global schema G. Now, given a query Q to G, one can generate a *query plan* that extracts the information from the sources [54,75,77,76]. In the global-as-view approach [73], the global database is defined as a view over the data sources.

Sometimes one assumes that certain integrity constraints hold in the global system, and those integrity constraints are used in the generation of the query plan; actually, there are situations where without integrity constraints no query plan can be generated [38,55,59]. The problem is that we cannot be sure that such global integrity constraints hold. Even in the presence of consistent data sources, a global system that integrates them may become

inconsistent. The global integrity constraints are not maintained and could easily be violated. In consequence, data integration is a natural scenario to apply the methodologies presented before. What we have to do is to retrieve consistent information from the global system.

Several new interesting issues appear, among them: (a) What is a consistent answer in this context? (b) If we are going to base this notion on a notion of repair, what is a repair? Notice that we do not have global *instances* to repair. (c) How can the consistent answers be retrieved from the global systems? What kind of query plans do we need? These and other issues are addressed in [18] for the local-as-view approach and in [72] for the global-as-view approach.

8.3 Other problems

An important achievement of this line of research would be to integrate the mechanisms for the retrieval of consistent query answers with a full-fledged DBMS. In such a system it should be possible to specify in SQL *soft* integrity constraints (constraints that are not explicitly maintained) and pose the usual SQL queries. The consistent answers to those queries would be obtained by an enhanced SQL engine. Note that different users, having different perceptions, could specify different soft constraints.

So far we have developed our notions of consistent answer and repair in the context of relational databases. Nevertheless, it would be interesting to extend these notions, and the corresponding computational mechanisms, to other forms of databases, like semi-structured, deductive, spatio/temporal etc.

9 Acknowledgments

This paper is dedicated to the memory of Ray Reiter whose pioneering work on logical foundations of databases laid the foundation for and influenced many research efforts in that area, including our own.

The contributions of the following researchers are gratefully acknowledged: Marcelo Arenas, Pablo Barcelo, Alexander Celle, Alvaro Cortes, Claudio Gutierrez, Xin He, Michael Kifer, Jerzy Marcinkowski, Francisco Orchard, Vijay Raghavan, Camilla Schwind, and Jeremy Spinrad. The anonymous reviewers are thanked for their detailed and helpful comments.

Research supported by FONDECYT Grant 1000593, joint NSF Grant INT-9901877/CONICYT Grant 1998-02-083, NSF Grant IIS-0119186, Carleton University Start-Up Grant 9364-01, and NSERC Grant 250279-02.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

2. S. Agarwal, A. M. Keller, G. Wiederhold, and K. Saraswat. Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. In *IEEE International Conference on Data Engineering*, 1995.
3. M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *ACM Symposium on Principles of Database Systems*, pages 68–79, 1999.
4. M. Arenas, L. Bertossi, and J. Chomicki. Specifying and Querying Database Repairs Using Logic Programs with Exceptions. In *International Conference on Flexible Query Answering Systems*, pages 27–41. Springer-Verlag, 2000.
5. M. Arenas, L. Bertossi, and J. Chomicki. Scalar Aggregation in FD-Inconsistent Databases. In *International Conference on Database Theory*, pages 39–53. Springer-Verlag, LNCS 1973, 2001.
6. M. Arenas, L. Bertossi, and J. Chomicki. Answer Sets for Consistent Query Answering in Inconsistent Databases. Technical Report arXiv:cs.DB/0204094, arXiv.org e-Print archive, July 2002. Under revision for journal publication.
7. M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar Aggregation in Inconsistent Databases. *Theoretical Computer Science*, 2003. Special issue: selected papers from ICDT 2001, to appear.
8. M. Arenas, L. Bertossi, and M. Kifer. Applications of Annotated Predicate Calculus to Querying Inconsistent Databases. In *International Conference on Computational Logic*, pages 926–941. Springer-Verlag, LNCS 1861, 2000.
9. O. Arieli and A. Avron. A Model-Theoretic Approach for Recovering Consistent Data from Inconsistent Knowledge Bases. *Journal of Automated Reasoning*, 22(2):263–309, 1999.
10. O. Arieli, M. Denecker, B. Van Nuffelen, and M. Bruynooghe. Repairing Inconsistent Databases: A Model-Theoretic Approach. In Decker et al. [36].
11. P. Asirelli, P. Inverardi, and G. Plagenza. Integrity Constraints as Views in Deductive Databases. In S. Conrad, H-J. Klein, and K-D. Schewe, editors, *International Workshop on Foundations of Models and Languages for Data and Objects*, pages 133–140, September 1996.
12. B. Balzer. Tolerating Inconsistency. In *International Conference on Software Engineering*, pages 158–165. IEEE Computer Society Press, 1991.
13. C. Baral, S. Kraus, J. Minker, and V. S. Subrahmanian. Combining Knowledge Bases Consisting of First-Order Theories. *Computational Intelligence*, 8:45–71, 1992.
14. P. Barcelo and L. Bertossi. Repairing Databases with Annotated Predicate Logic. In S. Benferhat and E. Giunchiglia, editors, *Ninth International Workshop on Non-Monotonic Reasoning (NMR02), Special Session: Changing and Integrating Information: From Theory to Practice.*, pages 160–170, 2002.
15. P. Barcelo and L. Bertossi. Logic Programs for Querying Inconsistent Databases. In *International Symposium on Practical Aspects of Declarative Languages*. Springer-Verlag, 2003. To appear.
16. R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
17. L. Bertossi and J. Chomicki, editors. *Working Notes of the IJCAI'01 Workshop on Inconsistency in Data and Knowledge*. AAAI Press, 2001.
18. L. Bertossi, J. Chomicki, A. Cortes, and C. Gutierrez. Consistent Answers from Integrated Data Sources. In *International Conference on Flexible Query Answering Systems*, Copenhagen, Denmark, October 2002. Springer-Verlag.

19. L. Bertossi and C. Schwind. Analytic Tableaux and Database Repairs: Foundations. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 32–48. Springer-Verlag, LNCS 2284, 2002.
20. L. Bertossi and C. Schwind. Database Repairs and Analytic Tableaux. Technical Report arXiv:cs.DB/0211042, arXiv.org e-Print archive, November 2002.
21. H. Blair and V. S. Subrahmanian. Paraconsistent Logic Programming. *Theoretical Computer Science*, 68(2):135–154, 1989.
22. A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: A Survey*. SIAM, 1999.
23. F. Bry. Query Answering in Information Systems with Integrity Constraints. In *IFIP WG 11.5 Working Conference on Integrity and Control in Information Systems*. Chapman &Hall, 1997.
24. A. Celle and L. Bertossi. Querying Inconsistent Databases: Algorithms and Implementation. In *International Conference on Computational Logic*, pages 942–956. Springer-Verlag, LNCS 1861, 2000.
25. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems*, 15(2):162–207, 1990.
26. A. Chandra and P. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. In *ACM SIGACT Symposium on the Theory of Computing*, pages 77–90, 1977.
27. L. Cholvy. Querying an Inconsistent Database. In *Proceedings of Artificial Intelligence : Methodology, Systems and Applications (AIMSA)*. North-Holland, 1990.
28. J. Chomicki and J. Marcinkowski. Minimal-Change Integrity Maintenance Using Tuple Deletions. Technical Report cs.DB/0212004, arXiv.org e-Print archive, December 2002.
29. J. Chomicki and J. Marcinkowski. On the Computational Complexity of Consistent Query Answers. Technical Report arXiv:cs.DB/0204010, arXiv.org e-Print archive, April 2002.
30. T. Chou and M. Winslett. A Model-Based Belief Revision System. *Journal of Automated Reasoning*, 12:157–208, 1994.
31. N. C. A. da Costa, J-Y. Beziau, and O. Bueno. Paraconsistent Logic in a Historical Perspective. *Logique & Analyse*, 150/151/152:111–125, 1995.
32. M. Dalal. Investigations into a Theory of Knowledge Base Revision. In *National Conference on Artificial Intelligence*, St.Paul, Minnesota, August 1988.
33. C. V. Damasio and L. M. Pereira. A Survey of Paraconsistent Semantics for Extended Logic Programs. In Gabbay and Smets [43], pages 241–320.
34. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
35. S. de Amo, W. A. Carnielli, and J. Marcos. A Logical Framework for Integrating Inconsistent Information in Multiple Databases. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 67–84, 2002.
36. H. Decker, J. Villadsen, and T. Waragai, editors. *ICLP 2002 Workshop on Paraconsistent Computational Logic (PCL)*, July 2002.
37. Phan Minh Dung. Integrating Data from Possibly Inconsistent Databases. In *International Conference on Cooperative Information Systems*, Brussels, Belgium, 1996.
38. O.M. Duschka, M.R. Genesereth, and A.Y. Levy. Recursive Query Plans for Data Integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

39. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving in DLV. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer, 2000.
40. T. Eiter and G. Gottlob. On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artificial Intelligence*, 57(2-3):227–270, 1992.
41. R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *ACM Symposium on Principles of Database Systems*, 1983.
42. E. Franconi, A. L. Palma, N. Leone, S. Perri, and F. Scarcello. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 561–578. Springer-Verlag, LNCS 2250, 2002.
43. D.M. Gabbay and P. Smets, editors. *Handbook of Defeasible Reasoning and Uncertain Information*, volume 2. Kluwer, 1998.
44. H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C-A. Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *International Conference on Very Large Data Bases*, pages 371–380, 2001.
45. P. Gärdenfors. *Knowledge in Flux*. Bradford Books, 1990.
46. P. Gärdenfors and H. Rott. Belief Revision. In D. M. Gabbay, J. Hogger, C, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, pages 35–132. Oxford University Press, 1995.
47. M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.
48. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–386, 1991.
49. M. Gertz. A Extensible Framework for Repairing Constraint Violations. In S. Conrad, H-J. Klein, and K-D. Schewe, editors, *International Workshop on Foundations of Models and Languages for Data and Objects*, pages 41–56, September 1996.
50. M. Gertz. *Diagnosis and Repair of Constraint Violations in Database Systems*. PhD thesis, Universität Hannover, 1996.
51. F. Giannotti, S. Greco, D. Sacca, and C. Zaniolo. Programming with Nondeterminism in Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 19(3-4), 1997.
52. F. Giannotti and D. Pedreschi. Datalog with Non-deterministic Choice Computes NDB-PTIME. *Journal of Logic Programming*, 35:75–101, 1998.
53. P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity Constraints: Semantics and Applications. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 9. Kluwer Academic Publishers, Boston, 1998.
54. G. Grahne and A. O. Mendelzon. Tableau Techniques for Querying Information Sources through Global Schemas. In *International Conference on Database Theory*, pages 332–347. Springer-Verlag, LNCS 1540, 1999.
55. J. Grant and J. Minker. A Logic-Based Approach to Data Integration. *Theory and Practice of Logic Programming*, 2(3):323–368, 2002.
56. G. Greco, S. Greco, and E. Zumpano. A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases. In *International Conference on Logic Programming*, pages 348–364. Springer-Verlag, LNCS 2237, 2001.

57. S. Greco, D. Sacca, and C. Zaniolo. Datalog Queries with Stratified Negation and Choice: from P to D^P. In *International Conference on Database Theory*, pages 82–96. Springer-Verlag, 1995.
58. S. Greco and E. Zumpano. Querying Inconsistent Databases. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 308–325. Springer-Verlag, LNCS 1955, 2000.
59. J. Gryz. Query Rewriting using Views in the Presence of Functional and Inclusion Dependencies. *Information Systems*, 24(7):597–612, 1999.
60. D. S. Hochbaum. Approximating Covering and Packing Problems: Set Cover, Vertex Cover, Independent Set, and Related Problems. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Co., 1997.
61. A. Hunter. Paraconsistent Logics. In Gabbay and Smets [43], pages 13–44.
62. T. Imieliński, S. Naqvi, and K. Vadaparty. Incomplete Objects - A Data Model for Design and Planning Applications. In *ACM SIGMOD International Conference on Management of Data*, pages 288–297, Denver, Colorado, May 1991.
63. T. Imieliński, R. van der Meyden, and K. Vadaparty. Complexity Tailored Design: A New Design Methodology for Databases With Incomplete Information. *Journal of Computer and System Sciences*, 51(3):405–432, 1995.
64. C. Kakas, A, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.
65. P. C. Kanellakis. Elements of Relational Database Theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 17, pages 1073–1158. Elsevier/MIT Press, 1990.
66. H. Katsuno and A. O. Mendelzon. Propositional Knowledge Base Revision and Minimal Change. *Artificial Intelligence*, 52(3):263–294, 1992.
67. M. Kifer and E. L. Lozinskii. A Logic for Reasoning with Inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, 1992.
68. M. Kifer and V. S. Subrahmanian. Theory of Generalized Annotated Logic Programming and its Applications. *Journal of Logic Programming*, 12(4):335–368, 1992.
69. R. Kowalski and F. Sadri. Logic Programs with Exceptions. *New Generation Computing*, 9(3/4):387–400, 1991.
70. R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *International Conference on Very Large Data Bases*, pages 61–69. Morgan Kaufmann Publishers, 1987.
71. S. M. Leach and J.J. Lu. Query Processing in Annotated Logic Programming: Theory and Implementation. *Journal of Intelligent Information Systems*, 6:33–58, 1996.
72. D. Lembo, M. Lenzerini, and R. Rosati. Source Inconsistency and Incompleteness in Data Integration. In *Workshop on Nonmonotonic Reasoning (NMR'02)*, Toulouse, France, 2002.
73. M. Lenzerini. Data Integration: A Theoretical Perspective. In *ACM Symposium on Principles of Database Systems*, 2002. Invited talk.
74. N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. *Information and Computation*, 135(2):69–112, 1997.
75. A. Y. Levy. Combining Artificial Intelligence and Databases for Data Integration. In *Artificial Intelligence Today*, pages 249–268. Springer-Verlag, LNCS 1600, 1999.

76. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Query-Answering Algorithms for Information Agents. In *National Conference on Artificial Intelligence*, pages 40–47, 1996.
77. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *International Conference on Very Large Data Bases*, pages 251–262, 1996.
78. J. Lin. A Semantics for Reasoning Consistently in the Presence of Inconsistency. *Artificial Intelligence*, 86(1-2):75–95, 1996.
79. J. Lin and A. O. Mendelzon. Merging Databases under Constraints. *International Journal of Cooperative Information Systems*, 7(1):55–76, 1996.
80. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
81. E. L. Lozinskii. Resolving Contradictions: A Plausible Semantics for Inconsistent Systems. *Journal of Automated Reasoning*, 12(1):1–32, 1994.
82. V. W. Marek and M. Truszczynski. Revision Programming. *Theoretical Computer Science*, 190(2):241–277, 1998.
83. A. Motro. Multiplex: A Formal Model for Multidatabases and Its Implementation. In *International Workshop on Next Generation Information Technology and Systems*, pages 138–158. Springer-Verlag, LNCS 1649, 1999.
84. B. A. Nuseibeh, S. M. Easterbrook, and Russo A. Leveraging Inconsistency in Software Development. *IEEE Computer*, 33(4):24–29, 2000.
85. S. Pradhan. Semantics of Normal Logic Programs and Contested Information. In *IEEE Symposium on Logic in Computer Science*, pages 406–415, 1996.
86. S. Pradhan. *Reasoning with Conflicting Information in Artificial Intelligence and Database Theory*. PhD thesis, Department of Computer Science, University of Maryland, 2001.
87. R. Reiter. Towards a Logical Reconstruction of Relational Database Theory. In M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On Conceptual Modeling*, pages 191–233. Springer-Verlag, 1984.
88. R. Reiter. On Integrity Constraints. In *International Conference on Theoretical Aspects of Rationality and Knowledge*, pages 97–111, 1988.
89. R. Reiter. What Should A Database Know? In *ACM Symposium on Principles of Database Systems*, pages 302–304, 1988.
90. F. Sadri and R. Kowalski. A Theorem-Proving Approach to Database Integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 313–362. Morgan Kaufmann Publishers, 1988.
91. K. F. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM SIGMOD International Conference on Management of Data*, pages 442–453, 1994.
92. V. S. Subrahmanian. Amalgamating Knowledge Bases. *ACM Transactions on Database Systems*, 19(2):291–331, 1994.
93. J. D. Ullman. Information Integration Using Logical Views. In *International Conference on Database Theory*, pages 19–40. Springer-Verlag, LNCS 1186, 1997.
94. M. Y. Vardi. The Complexity of Relational Query Languages. In *ACM Symposium on Theory of Computing*, pages 137–146, 1982.
95. J. Wijsen. Condensed Representation of Database Repairs for Consistent Query Answering. In *International Conference on Database Theory*, 2003.
96. M. Winslett. Reasoning about Action using a Possible Models Approach. In *National Conference on Artificial Intelligence*, 1988.

Index