Minimizing Risks of Decision Making by Inconsistency-Tolerant Integrity Management

Hendrik Decker*

Instituto Tecnológico de Informática UPV,
Ciudad Politécnica de la Innovación, 46022 Valencia, Spain

Abstract. In practice, knowledge-based reasoning for decision making must be inconsistency-tolerant since, for decision making, contradictory data are unavoidable. We present a measure-based concept of inconsistency-tolerant knowledge engineering for decision support. It enables the preservation of consistency across updates, as well as the computation of sound answers to queries in knowledge bases with violated integrity. Hence, our framework supports the consistency of decision making in the presence of contradictory data. By an extended example, we show how inconsistency-tolerant integrity maintenance can minimize risks in decision making that result from inconsistent knowledge.

1 Introduction

In computational knowledge engineering, inconsistency tolerance is the capacity of sound information processing in the presence of inconsistent data [3]. Its main applications are query answering [7], database integration [13] and integrity maintenance [10]. The application of inconsistency tolerance proposed in this paper is knowledge engineering for decision support, which involves integrity checking, inconsistency repairing and query answering. The main contribution of this paper is to provide evidence of the feasibility of a measure-based concept of inconsistency-tolerant knowledge engineering for consistent decision support including the minimizations of risks resulting from contradictory data.

Nowadays, decision making in enterprises is supported by knowledge-based computing systems that provide for a rational analysis of large amounts of stored business data. The backbone of such knowledge-based systems are databases. They support the evaluation of complex queries, which is essential for proper decision making. Answers on which far-reaching decisions depend should be as consistent as possible, since otherwise, fatally wrong decisions could be taken.

Decision support systems should be able to work in the presence of inconsistent data, since they are unavoidable and may even be useful, e.g., for finding optimal tradeoffs between contradictory goals. Thus, query answering should provide reasonable answers even if integrity is violated. Similarly, integrity checking should make reasonable decisions to accept or reject updates in the presence of inconsistencies. In fact, sound reasoning with unsound data is a big challenge for

* Supported by ERDF/FEDER and MECgrants TIN2009-14460-C03, TIN2010-17139, TIN2012-37719-C03-01.

Y.T. Demey and H. Panetto (Eds.): OTM 2013 Workshops, LNCS 8186, pp. 556–565, 2013.

database logic, which is based on classical predicate logic. The latter does not survive contradictions [18], predicting that everything, thus nothing valuable, can be inferred as an answer to a query in an inconsistent database.

Section 2 illustrates how semantic consistency as well as properties of potential risks can be captured in the syntax of integrity constraints. In Section 3, inconsistency-tolerant integrity checking (abbr. *ITIC*) is shown to prevent a deterioration of decision making. In Section 4, *ITIC* is used also for repairing damaged data, without insisting on a total elimination of all violations. In Section 5, we sketch how valid answers to queries for decision making can be obtained from knowledge bases that contain risky or inconsistent data. In Section 6, we feature an example of managing risky data, with comparisons to several alternative approaches. In Section 7, we address related work. In Section 8, we conclude.

Terminology and notations in this paper come from deductive databases [1]. Throughout, symbols like D, I, IC, U always stand for a database, an integrity constraint, a finite set of constraints (also called *integrity theory*) and, resp., an update. The result of updating D by U is denoted by D^U, and $D(S)$ is the result of evaluating a set of sentences S in D.

2 Modeling Risk as Constraints

Integrity constraints (in short: constraints) in databases are asserted as logical sentences and often as *denials*, i.e., clauses of the form $\leftarrow B$, where the body B states what *should not* be true. Thus, each instance of B that is true in the database indicates an integrity violation. We begin with two examples of classical integrity constraints, and then argue by further examples that properties for capturing risky data can be modeled in the same syntax.

In a database for decision making by medical staff,

$$\forall x \forall y \forall z (\mathit{myoglobin}(x, y, z) \rightarrow \mathit{person}(x) \wedge \mu\mathit{mol\text{-}per\text{-}l}(y) \wedge \mathit{time}(z))$$

stipulates that the first attribute of the *myoglobin* table always is a person, the second a micromoles/liter value and the third of type *date*. Similarly, the denial

$$\leftarrow \mathit{myoglobin}(x, y1, z),\ \mathit{myoglobin}(x, y2, z),\ y1 \neq y2$$

declares a primary key constraint on the first and third columns, preventing multiple entries for a person with different myoglobin values at the same time.

Likewise, conditions for characterizing data as risky can be modeled in the syntax of integrity constraints. For instance, consider $\leftarrow \mathit{risk}(x, z)$, where *risk* is defined by $\mathit{risk}(x, z) \leftarrow \mathit{person}(x)$, $\mathit{myoglobin}(x, y, z)$, $\mathit{above\text{-}threshold}(y)$, and $\mathit{above\text{-}threshold}(y)$ be a predicate that compares the value of y with a suitable constant boundary. Thus, the health of a person x at time z is considered to be at risk if the myoglobin level y exceeds some critical threshold.

The denial $\leftarrow \mathit{dubious}(x,y)$ states doubts (i.e., a risk of incorrectness) about the data in a municipal database for decision making by the mayor, where the predicate *dubious* is defined by $\mathit{dubious}(x, y) \leftarrow \mathit{birth\text{-}date}(x, y), y < 1900$. Thus, each entry of a person x with birth date z prior to the 20th century is considered dubious, i.e., it is risky to trust such data.

The following clause serves to define the constraint $\leftarrow risky(x)$, by disqualifying the trustworthiness of rows x in database tables y whose confidence value z is below some threshold th: $risky(x) \leftarrow confidence(row(x, y), z) \wedge z < th$.

The denial $\leftarrow suspect(x)$ models suspect and thus risky email, where *suspect* is defined by $suspect(x) \leftarrow email(x, from(y)), \sim authenticated(y)$, which warns that an email x sent by y is suspect or risky because it may contain malicious elements if y cannot be authenticated. Note that emails x that qualify as suspect may well be acceptable despite the risks caused by senders y.

Conventionally, updates that would violate any constraint are rejected. As opposed to that, the preceding examples illustrate that violations of constraints describing risks may be tolerable; some more, some less. In the remainder, we show that not only conventional integrity constraints, but also risk constraints can be maintained by inconsistency-tolerant methods of integrity management for checking the preservation of integrity by updates, for repairing violations, and for providing answers that have integrity in knowledge bases that don't.

3 Inconsistency-Tolerant Constraint Checking

Constraints are meant to be checked upon each update, which is committed only if it does not violate any constraint. The same can be done for risk constraints. However, a total absence of risks is hard to be maintained, and constraint violations may at times be acceptable anyway, as we have seen in Section 2. Thus, methods for checking constraints that can tolerate extant violations are needed.

In [11], we have shown that most (but not all) known integrity checking methods are inconsistency-tolerant, even though they have been designed to work only if integrity is totally satisfied before any update is checked. In [10], we have seen that most integrity checking methods can be described by *violation measures*. Each such measure maps pairs (D, IC) to some partially ordered space, for sizing the violated constraints in (D, IC). Thus, an update can be accepted if it does not increase the measured amount of constraint violations.

Definition 1 characterizes each constraint checking method $\mathcal{M}$ (in short, method) as an I/O function that maps input triples (D, IC, U) to $\{ok, ko\}$. The output *ok* means that the checked update is acceptable, and *ko* that it may not be acceptable. For deciding to *ok* or *ko* an update U, $\mathcal{M}$ uses a violation measure μ, the range of which is structured by some partial order $\preccurlyeq$, for determining if U increases the amount of measured violations or not.

Definition 1. (*Measure-based ITIC*)
An integrity checking method maps triples (D, IC, U) to $\{ok, ko\}$. For a violation measure $(\mu, \preccurlyeq)$, a method $\mathcal{M}$ is *sound* (resp., *complete*) for μ-based integrity checking if, for each (D, IC, U), (1) (resp., (2)) holds.

$$\mathcal{M}(D, IC, U) = ok \quad \Rightarrow \quad \mu(D^U, IC) \preccurlyeq \mu(D, IC) \tag{1}$$

$$\mu(D^U, IC) \preccurlyeq \mu(D, IC) \quad \Rightarrow \quad \mathcal{M}(D, IC, U) = ok \tag{2}$$

The only real difference between conventional methods and integrity checking as defined above is that the former additionally requires total integrity before the update, i.e., that $D(IC) = \mathit{true}$ in the premise of Definition 1. The measure μ used by conventional methods is binary: $\mu(D, IC) = \mathit{true}$ means that IC is satisfied in D, and $\mu(D, IC) = \mathit{false}$ that it is violated.

As seen in [11], many conventional methods can be turned into sound (though not necessarily complete) inconsistency-tolerant ones, simply by waiving the premise $D(IC) = \mathit{true}$ and comparing violations in (D, IC) and (D^U, IC). If there are more or new violations in (D^U, IC) that are not in (D, IC), then they output *ko*; otherwise, they may output *ok*.

4 Inconsistency-Tolerant Repairs

Essentially, repairs are updates of databases that eliminate their constraint violations. However, the user or the application may not be aware of each violation, so that some of then may be missed when trying to repair a database.

Below, we recapitulate the definition of repairs in [11] which is inconsistency-tolerant since it permits that some violations may persist after a partial repair.

Definition 2. (*Repair*)
For a triple (D, IC, U), let S be a subset of IC such that $D(S) = \mathit{false}$. An update U is called a *repair* of S in D if $D^U(S) = \mathit{true}$. If $D^U(IC) = \mathit{false}$, U is also called a *partial repair* of IC in D. Otherwise, if $D^U(IC) = \mathit{true}$, U is called a *total repair* of IC in D.

Unfortunately, partial repairs may inadvertently cause the violation of some constraint that is not in the repaired subset.

Example 1. Let $D = \{p(1,2,3),\ p(2,2,3),\ p(3,2,3),\ q(1,3), q(3,2), q(3,3)\}$ and $IC = \{\leftarrow p(x,y,z) \wedge \sim q(x,z),\ \leftarrow q(x,x)\}$. Clearly, both constraints are violated. $U = \{\mathit{delete}\ q(3,3)\}$ is a repair of $\{\leftarrow q(3,3)\}$ in D and hence a partial repair of IC. It tolerates the persistence of the violation $\leftarrow p(2,2,3) \wedge \sim q(2,3)$ in D^U. However, U also causes the violation $\leftarrow p(3,2,3) \wedge \sim q(3,3)$ of the first constraint of IC in D^U. That instance is not violated in D. Thus, the non-minimal partial repair $U' = \{\mathit{delete}\ q(3,3),\ \mathit{delete}\ p(3,2,3)\}$ is needed to eliminate the violation of $\leftarrow q(3,3)$ in D without causing a violation that did not yet exist.

Although U' does not cause any unpleasant side effect as U does, such repair iterations may in general continue indefinitely, as known from repairing by triggers [5]. However, that can be alleviated by checking if a given repair is an update that *preserves integrity*, i.e., does not increase the amount of violations, with any measure-based method that prevents inconsistency from increasing while tolerating extant constraint violations. Hence, we have the following result.

Theorem [10] Let μ be a violation measure, $\mathcal{M}$ a μ-based integrity checking method and U a partial repair of IC in D. For a tuple (D, IC), U preserves integrity wrt. μ if $\mathcal{M}(D, IC, U) = \mathit{ok}$.

For computing partial repairs, any off-the-shelve view update method can be used, as follows. Let $S = \{\leftarrow B_1, \ldots, \leftarrow B_n\}$ be a subset of constraints to be repaired in the database D of an information system. Candidate updates for satisfying the view update request can be obtained by running the view update request *delete violated* in $D \cup \{violated \leftarrow B_i \mid 0 \leq i \leq n\}$. For deciding if a candidate update U is a valid repair, U can be checked for integrity preservation by some measure-based method, according to the preceding theorem. More details about the computation of partial and total repairs can be found in [10].

5 Answers That Tolerate Violations of Constraints

Violated risk constraints may impair the validity of query answering, since the data that provide the answers are precarious. Thus, an approach to provide answers with integrity in knowledge bases with violated constraints is needed.

Consistent query answering (abbr. *CQA*) [2] provides answers that are true in each minimal total repair of *IC* in D. CQA uses semantic query optimization [6] which in turn uses integrity constraints for query answering. A similar approach is to abduce consistent hypothetical answers, together with a set of hypothetical updates that can be interpreted as integrity-preserving repairs [12].

A new approach to provide answers that have integrity (abbr. AHI), and a comparison to CQA, is presented in [9]. AHI determines two sets of data: those by which an answer is deduced, i.e., the causes of the answer, and those that cause constraint violations. Each cause is a set of ground instances of if- and only-if halves of predicate completions in $comp(D)$ [8]. An answer θ has integrity if the intersection of one of the causes of the answer with the causes of constraint violations is empty, since θ is deducible from data that are independent of those that violate constraints.

AHI is closely related to measure-based *ITIC*, since some convenient violation measures are defined by causes: cause-based methods accept an update U only if U does not increase the number or the set of causes of constraint violations [10]. Similar to *ITIC*, AHI is inconsistency-tolerant since it provides correct results in the presence of constraint violations. However, AHI is not as inconsistency-tolerant as measure-based *ITIC*, since each answer accepted by AHI is independent of inconsistent parts of the database, while measure-based *ITIC* may admit updates that violate constraints. For instance, U in Example 1 causes the violation of a constraint while eliminating some other violation. Now, if U is checked by a method based on a measure that assigns a greater weight to the eliminated violation than to the newly caused one, U will be *ok*-ed, because it decreases the measured amount of inconsistency.

In fact, it should be possible to provide answers that, despite some tolerable degree of contamination with inconsistency, are appreciable. The idea is to provide answers that tolerate a certain amount of violations of constraints that may be involved in the derivation of answers. To quantify that amount, some application-specific tolerance measure is needed. In ongoing research, we elaborate a theory based on that idea.

6 Risk Management – An Example

Risky data often cannot be totally avoided. Hence the desire to contain or reduce risky data, so that their amount does not grow and won't compromise the validity of answers too much. In this section, we illustrate how to meet that desideratum by inconsistency-tolerant integrity management, and discuss some more conventional alternatives. In particular, we compare inconsistency-tolerant integrity management with brute-force constraint evaluation, conventional integrity checking that is not inconsistency-tolerant, total repairing, and CQA, in 6.1 – 6.6.

The example below is open to interpretation. By assigning convenient meanings to predicates, it can be interpreted as a model of risky data in a decision support systems for, e.g., stock trading, or controlling operational hazards in a complex system.

Let D be a database with the following definitions of view predicates *rl*, *rm*, *rh* that model risks of low, medium and, respectively, high degree:

$rl(x) \leftarrow p(x,x)$

$rm(y) \leftarrow q(x,y), \sim p(y,x)$; $\quad rm(y) \leftarrow p(x,y), q(y,z), \sim p(y,z), \sim q(z,x)$

$rh(z) \leftarrow p(0,y), q(y,z), z > th$

where *th* be a threshold value that always is greater or equal 0. Now, let risk be denied by the following integrity theory:

$IC \;=\; \{\leftarrow rl(x), \;\; \leftarrow rm(x), \;\; \leftarrow rh(x)\}.$

Note that IC is satisfiable, e.g., by $D = \{p(1,2), p(2,1), q(2,1)\}$. Now, let the extensions of p and q in D be populated as follows.

$p(0,0), p(0,1), p(0,2), p(0,3), \ldots, p(0,1000000),$

$p(1,2), p(2,4), p(3,6), p(4,8), \ldots, p(500000,1000000)$

$q(0,0), q(1,0), q(3,0), q(5,0), q(7,0), \ldots, q(999999,0)$

It is easy to verify that the low-risk denial $\leftarrow p(x,x)$ is the only constraint that is violated in D, and that this violation is caused by $p(0,0)$.

Now, let us consider the update $U = \mathit{insert}\ q(0,999999)$.

6.1 Brute-force Risk Management

For later comparison, let us first analyze the general cost of brute-force evaluation of IC in D^U. Evaluating $\leftarrow rl(x)$ involves a full scan of p. Evaluating $\leftarrow rm(x)$ involves access to the whole extension of q, a join of p with q, and possibly many lookups in p and q for testing the negative literals. Evaluating $\leftarrow rh(x)$ involves a join of p with q plus the evaluation of possibly many ground instances of $z > th$.

For large extensions of p and q, brute-force evaluation of IC clearly may last too long, in particular for safety-critical risk monitoring in real time. In 6.2, we are going to see that it is far less costly to use an ITIC method that simplifies the evaluation of constraints by confining its focus on the data that are relevant for the update.

6.2 Inconsistency-Tolerant Risk Management

First of all, note that the use of conventional simplification methods that require the satisfaction of IC in D is not allowed in our example, since $D(IC) = \mathit{false}$. Thus, conventional integrity checking has to resort on brute-force constraint evaluation. We are going to see that inconsistency-tolerant integrity checking of U is much less expensive than brute-force evaluation.

At update time, the following three simplifications of medium and high risk constraints are obtained from U. (No low risk can be caused by U since $q(0, 999999)$ does not match $p(x, x)$.) These simplifications are obtained at hardly any cost, by simple pattern matching of U with pre-simplified constraints that can be compiled at constraint specification time.

$\leftarrow \sim p(999999,0);\ \leftarrow p(x,0), \sim p(0,999999), \sim q(999999,x);\ \leftarrow p(0,0), 999999 > th$

By a simple lookup of $p(999999, 0)$ for evaluating the first of the three denials, it is inferred that $\leftarrow rm(x)$ is violated.

Now that a medium risk has been spotted, there is no need to check the other two simplifications. Yet, let us do that, for later comparison in 6.3.

Left-to-right evaluation of the second simplification essentially equals the cost of computing the answer $x = 0$ to the query $\leftarrow p(x, 0)$ and successfully looking up $q(999999, 0)$. Hence, the second denial is *true*, which means that there is no further medium risk. Clearly, the third simplification is violated if $999999 > th$ is true, since $p(0, 0)$ is true, i.e., there possibly is a high risk.

Now, let us summarize this subsection. Inconsistency-tolerant integrity checking of U essentially costs a simple access to the p relation. Only one more lookup is needed for evaluating all constraints. And, apart from a significant cost reduction, ITIC prevents medium and high risk constraint violations that would be caused by U if it were not rejected.

6.3 Inconsistency-Intolerant Risk Management

ITIC is logically correct, but, in general, methods that are not inconsistency-tolerant (e.g., those in [14,16]) are incorrect, as shown by the example below.

Clearly, p is not affected by U. Thus, $D(\leftarrow rl(x)) = D^U(\leftarrow rl(x))$. Recall that each method that is not inconsistency-tolerant assumes $D(IC) = \mathit{true}$. Thus, such methods would wrongly conclude that the unfolding $\leftarrow p(x, x)$ of $\leftarrow rl(x)$ is satisfied in D and D^U, although $p(0, 0) \in D$. That conclusion is then applied to $\leftarrow p(0, 0),\ 999999 > th$, (the third of the simplifications in 6.2), which thus is taken to be satisfied in D^U. That, however, is wrong if $999999 > th$ is true. Thus, non-inconsistency-tolerant integrity checking may wrongly infer that the high risk constraint $\leftarrow rh(z)$ cannot be violated in D^U.

6.4 Risk Management by Repairing (D, IC)

Conventional integrity checking requires $D(IC) = \mathit{true}$. To comply with that, all violations in (D, IC) must be repaired before each update. However, such repairs can be exceedingly costly, as argued below.

In fact, already the identification of all violations in (D, IC) at update time may be prohibitively costly. But there is only a single low risk constraint violation in our example: $p(0,0)$ is the only cause of the violation $\leftarrow$ *rl(0)* in D. Thus, to begin with repairing D means to request $U = \mathit{delete}\ p(0,0)$, and to execute U if it preserves all constraints, according to the theorem in Section 4.

To check U for integrity preservation means to evaluate the simplifications

$\leftarrow q(0,0)$ and $\leftarrow p(x,0),\ q(0,0),\ {\sim}q(0,x)$

i.e., the two resolvents of $\sim p(0,0)$ and the clauses defining *rm*, since U affects no other constraints. The second one is satisfied in D^U, since there is no fact matching $p(x,0)$ in D^U. However, the first one is violated, since $D^U(q(0,0)) = \mathit{true}$. Hence, also $q(0,0)$ must be deleted. That deletion affects the constraint

$rm(y) \leftarrow p(x,y),\ q(y,z),\ {\sim}p(y,z),\ {\sim}q(z,x)$

and yields the simplification

$\leftarrow p(0,y),\ q(y,0),\ {\sim}p(y,0).$

As is easily seen, this simplification is violated by each pair of facts of the form $p(0,o)$, $q(o,0)$ in D, where o is an odd number in $[1, 999999]$. Thus, deleting $q(0,0)$ for repairing the violation caused by deleting $p(0,0)$ causes the violation of each instance of the form $\leftarrow rm(o)$, for each odd number o in $[1, 999999]$.

Hence, repairing each of these instances would mean to request the deletion of many rows of p or q. We shall not further track those deletions, since it should be clear already that repairing D is complex and tends to be significantly more costly than ITIC. Another advantage of ITIC: since inconsistency can be temporarily tolerated, ITIC-based repairs do not have to be done at update time. Rather, they can be done off-line, at any convenient point of time.

6.5 Risk Management by Repairing (D^U, IC)

Similar to repairing (D, IC), repairing (D^U, IC) also is more expensive than to tolerate extant constraint violations until they can be repaired at some more convenient time. That can be illustrated by the three violations in D^U, as identified in 6.1 and 6.2: the low risk that already exists in D, and the medium and high risks caused by U and detected by ITIC. To repair them obviously is even more intricate than to only repair the first of them as tracked in 6.4.

Moreover, for risk management in safety-critical applications, it is no good idea to simply accept an update without checking for potential violations of constraints, and to attempt repairs only after the update is committed, since repairing takes time, during which an updated but unchecked state may contain possibly very dangerous risks of any order.

6.6 Reliable Constraint Evaluation in Risky Databases

As already mentioned in Section 5, CQA is an approach to cope with constraint violations for query evaluation. There is a clear kinship of ITIC and CQA, since checking and repairing risk constraints involves their evaluation. However, the

evaluation of constraints by CQA is unprofitable, since consistent answers are defined to be those that are true in each minimally repaired database. Thus, by definition, CQA returns the empty answer for each queried denial constraint I, indicating the satisfaction of I. Thus, answers to queried constraints computed by CQA have no meaningful interpretation.

For example, CQA computes the empty answer to the query $\leftarrow rl(x)$ and to $\leftarrow rh(z)$, for any extension of p and q. However, the only reasonable answers to $\leftarrow rl(x)$ and $\leftarrow rh(z)$ in D are $x = 0$ and, resp., $x = 999999$, if $999999 > th$. These answers correctly indicate low and high risks in D and, resp., D^U.

For computing correct answers to queries (rather than to denials representing constraints), AHI is a viable alternative to CQA. A comparison, which turned out to be advantageous for AHI, has been presented in [9].

7 Related Work

Although integrity, i.e., conditions of semantic consistency, and properties that capture risks of some sort are obviously related, it seems that they never have been approached in a uniform way, as in this paper, neither in theory nor in practice. However, similarities and differences between the integrity of data and data that capture properties such as risks are identified in a collection of work on modeling and managing uncertain data [17]. In that book, largely diverse proposals to handle data that lack quality, and hence may involve some risk, are discussed. In particular, approaches such as probabilistic and fuzzy set modeling, exception handling, repairing and paraconsistent reasoning are dealt with. However, no particular approach to integrity checking is considered.

Several paraconsistent logics that tolerate inconsistency of data have been proposed, e.g., in [3,4]. Each of them departs from classical first-order logic, by adopting some annotated, probabilistic, modal or multivalued logic, or by replacing standard axioms and inference rules with non-standard axiomatizations. As opposed to that, inconsistency-tolerant integrity checking fully conforms with standard datalog and does not need any extension of classical logic.

Work on semantic inconsistencies in databases is also done in the field of measuring logical inconsistency [15]. Our violation measures also work in databases with non-monotonic negation, whereas inconsistency measures in [15]. do not deal with non-monotonicity, as argued in [10].

8 Conclusion

We have presented a non-standard approach to risk management. We have shown that risk can be modeled by integrity constraints. It can be contained and reduced by conventional integrity maintenance methods that are inconsistency-tolerant. From [10], we have adopted a generic description of inconsistency-tolerant methods on the basis of violation measures. Such measures also enable an evaluation of queries in knowledge bases for decision making such that the computed answers are not contaminated by risk-inflicted data. As illustrated in

Section 6, the use of inconsistency-tolerant approaches is essential, since wrong, possibly fatal decisions can be inferred from deficient data by methods that are not inconsistency-tolerant.

Apart from the investigations mentioned in Section 5, ongoing work is dedicated to scale up the results of this paper to maintaining constraints across concurrent transactions in distributed and replicated databases.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proc. 18th PODS, pp. 68–79. ACM Press (1999)
3. Bertossi, L., Hunter, A., Schaub, T. (eds.): Inconsistency Tolerance. LNCS, vol. 3300. Springer, Heidelberg (2005)
4. Carnielli, W., Coniglio, M., D'Ottaviano, I. (eds.): The Many Sides of Logic. Studies in Logic, vol. 21. College Publications, London (2009)
5. Ceri, S., Cochrane, R., Widom, J.: Practical Applications of Triggers and Constraints: Success and Lingering Issues. In: 26th VLDB, pp. 254–262. Morgan Kaufmann (2000)
6. Chakravarthy, U., Grant, J., Minker, J.: Logic-based approach to semantic query optimization. Transactions on Database Systems 15(2), 162–207 (1990)
7. Chomicki, J.: Consistent query answering: Five easy pieces. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. 4353, pp. 1–17. Springer, Heidelberg (2006)
8. Clark, K.: Negation as Failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press (1978)
9. Decker, H.: Answers That Have Integrity. In: Schewe, K.-D. (ed.) SDKB 2010. LNCS, vol. 6834, pp. 54–72. Springer, Heidelberg (2011)
10. Decker, H.: Measure-Based Inconsistency-Tolerant Maintenance of Database Integrity. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2013. LNCS, vol. 7693, pp. 149–173. Springer, Heidelberg (2013)
11. Decker, H., Martinenghi, D.: Inconsistency-tolerant Integrity Checking. TKDE 23(2), 218–234 (2011)
12. Fung, T.H., Kowalski, R.: The IFF proof procedure for abductive logic programming. J. Logic Programming 33(2), 151–165 (1997)
13. Fuxmann, A., Miller, R.: Towards Inconsistency Management in Data Integration Systems. In: Proc. IJCAI 2003 Workshop IIWEB, pp. 143–148 (2003)
14. Gupta, A., Sagiv, Y., Ullman, J., Widom, J.: Constraint checking with partial information. In: Proc. 13th PODS, pp. 45–55. ACM Press (1994)
15. Hunter, A., Konieczny, S.: Approaches to measuring inconsistent information. In: Bertossi, L., Hunter, A., Schaub, T. (eds.) Inconsistency Tolerance. LNCS, vol. 3300, pp. 191–236. Springer, Heidelberg (2005)
16. Lee, S.Y., Ling, T.W.: Further improvements on integrity constraint checking for stratifiable deductive databases. In: 22th International Conference on Very Large Data Bases, pp. 495–505. Morgan Kaufmann (1996)
17. Motro, A., Smets, P.: Uncertainty Management in Information Systems: From Needs to Solutions. Kluwer (1996)
18. Restall, G.: Laws of Non-contradiction, Laws of Excluded Middle and Logics. In: Priest, G., et al. (eds.) The Law of Non-Contradiction - New Philosophical Essays, pp. 73–85. Oxford University Press (2004)