MODELING AND MONITORING THE QUALITY OF DATA BY INTEGRITY CONSTRAINTS AND INTEGRITY CHECKING

Hendrik Decker
Instituto Tecnológico de Informática, Valencia, Spain

Keywords: Quality, Integrity constraints, Integrity checking, Inconsistency tolerance.

Abstract: Characteristic attributes for describing the quality of data, such as trustworthyness, soundness, riskiness, uncertainty, dependability, reliability and other semantic properties can be modeled and monitored by conventional database integrity technology. As opposed to traditional consistency constraints, occasional violations of some of the integrity conditions that describe quality aspects may be tolerable, even for extended periods of time. Traditional integrity checking methods are intolerant wrt. any constraint violation. They insist that all constraints are totally satisfied before updates can be checked for integrity preservation. Inconsistency-tolerant methods can waive that insistence. Thus, if data quality is modeled by constraints, it can be monitored by any integrity checking method that is inconsistency-tolerant. We illustrate that by an extended example, by which inconsistency-tolerant integrity checking is also compared to some alternative aproaches.

1 INTRODUCTION

In relational databases, first-order predicate logic sentences called *assertions*, *integrity constraints* or simply *constraints* are used to express conditions that are required to be invariantly satisfied across state changes caused by updates.

In knowlege bases and decision support systems, the expressive power of logic also can be used to capture any other semantic information that goes beyond the simple structures of common database content. In particular, conditions for characterising the quality of data can be expressed as assertions.

The basic motivation and idea behind this paper is that the database should not suffer from semantically imperfect data, and that stored data can be considered to have sufficient quality if the database satisfies suitably many quality constraints. Otherwise, if a critical amount of the required quality assertions is violated, then the quality of the database is impaired or damaged beyond tolerable proportions.

The evaluation of assertions tends to be prohibitively expensive. Thus, for the interpretation of assertions as integrity constraints, specific integrity checking methods for simplifying their evaluation are used. However, since the semantics of quality assertions is different from that of integrity constraints, the use of integrity checking methods for simplifying the evaluation of quality assertions is questionable.

Traditional integrity checking insists on total constraint satisfaction. That is not suitable in general for monitoring quality assertions, since some of them may be occasionally violated, even for extended periods of time, without impairing ongoing routine operations. As opposed to that, we are going to see that integrity checking methods that are able to tolerate extant violations of constraints also are able to monitor the dynamics of the quality of data.

More precisely, we show how to gain a better control over the quality and possible imperfections of stored data, by expressing quality properties as constraints, and monitoring them with inconsistency-tolerant integrity checking methods. Conditions that model quality properties may qualify data positively, e.g., as trustworthy, secure or healthy, or negatively, as imperfect or risky, uncertain or vague, etc. If each such property is satisfied, there is no quality erosion that would violate the constraints. Conversely, violation of quality properties means that the data that are responsible for violation qualify as corrupt.

Capturing quality properties of data by describing them in the form of integrity constraints yields a double benefit: Firstly, to use the expressive power of the syntax of semantic integrity constraints also for describing arbitrarily general quality properties of data. Secondly, to make use of established integrity check-

Decker H. (2009).
MODELING AND MONITORING THE QUALITY OF DATA BY INTEGRITY CONSTRAINTS AND INTEGRITY CHECKING.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 207-214
DOI: 10.5220/0002264602070214

ing methods in order to efficiently check data also for quality. Thus, monitoring and controlling stored and incoming new data and updates with regard to their quality can be achieved.

In section 2, we first strive to gain a better understanding of the similarities and differences between integrity and quality. Then, we claim that, in spite of seemingly severe differences, it is possible to capture conditions for quality by integrity constraints, and to monitor them by using methods for integrity checking. This claim is substantiated in the remainder. In section 3, we recapitulate the concept of inconsistency-tolerant integrity checking (Decker and Martinenghi, 2006; Decker, 2008). (Inconsistency here is synonymous to integrity violation.) We show that it is precisely the inconsistency tolerance of methods that makes them apt to be used for monitoring quality. In section 4, we elaborate an extended example that illustrates how inconsistency-tolerant integrity checking can be used for risk management. The latter is a special case of managing the quality of data. In section 5, we address related work. In section 6, we conclude.

2 QUALITY AND INTEGRITY

In 2.1, we analyse the similarities, and in 2.2 the differences between quality and integrity. Essentially, integrity and quality are very similar since both can be described by assertions. They differ since integrity constraints and their evaluation traditionally are much more exigent than quality assertions. However, as explained in 2.3, this difference is reconciled and can be overcome by inconsistency-tolerant methods for integrity checking, with which also quality assertions can be evaluated.

2.1 Similarities

Traditionally, integrity constraints are used to express correctness conditions with which all stored data must comply. Upon each issued update, the constraints imposed on the database are checked. Updates are committed only if they do not cause integrity violation. For example, in a civil registry database containing information about citizens and their marital status, inserting *married(john, mary)* violates

$$\forall x \forall y \forall z (married(x,y) \wedge married(x,z) \rightarrow y \neq z),$$

i.e., a constraint forbidding bigamy, if the tuple *married(john, susan)* is already stored. Then, also the integrity constraint

$$\forall x \forall y \; married(x,y) \rightarrow person(x,m) \wedge person(y,m)$$

which requires that each spouse of each married couple is registered in the *person* table of the database and has the marital status attribute set to *m(arried)*, will signal violation upon an attempt to delete the tuple $person(susan, m)$.

Also conditions for characterising database entries that lack quality (i.e., data that are risky, uncertain, precarious, dubious, suspicious, etc), as well data that are devoid of such deficiencies (i.e., data that are trustworthy, dependable, reliable, credible, etc) can be expressed in the syntax of integrity constraints. Obviously, *datalog* negation can be used to convert positive into negative qualities and vice-versa.

The constraint $uncertain \leftarrow person(x, null)$, for instance, qualifies each entry in the *person* table as uncertain by a namesake 0-ary predicate if the marital status of that entry is unknown, as represented by a *null* value. Another example is the constraint

$$\forall x \; p(x, null) \rightarrow p(x,s) \vee p(x,m) \vee p(x,d) \vee p(x,w)$$

where *p* abbreviates *person*. It says that each person with unknown marital status is either single or married or divorced or widowed. Similarly, entries of persons with birth date before the 20th century can be characterized as dubious. By amalgamating higher-order predicates into first-order terms (Bowen and Kowalski, 1982), also sentences such as

$$confidence(row(x,y),z) \wedge z < th \; \rightarrow \sim trustworthy(x)$$

may serve as constraints for disqualifying the trustworthyness of rows *x* in database tables *y* such that the confidence value *z* of *x* is below a certain threshold value *th* (which can be thought of as a constant parameter or the output of some evaluable function).

Thus, it is possible to characterize uncertain data in the syntax of integrity constraints. Analogously, this can also be done for any semantic properties that capture some other quality aspects. Hence, it should also be possible to use integrity checking methods in order to check incoming data for violations of quality constraints, and, symmetrically, check deletions for having the effect of impairing the quality of the remaining data. In the following subsection, we are going to see that this is not as straightforward as it may seem at first glace.

2.2 Differences

In 2.1, we have seen that the representation of properties describing logical consistency or quality is very similar. Both can be modeled by integrity assertions.

However, there is a significant difference between quality and integrity. Data that lack integrity are not just uncertain or dubious, but definitely bad, while data the quality of which is compromised may or

may not have integrity. Essentially, the difference is that integrity is two-valued (i.e., satisfied or violated), while quality is not binary, or at least not in the sense that imperfect data with impaired quality would always be invalid. In fact, data that violate integrity are usually considered useless, harmful and unwanted, while data that lack quality can still be useful.

For example, the integrity constraint

$$violated \leftarrow emp(x),\ age(x,y),\ y < 14$$

expresses that integrity is violated by underage employment (because there is a law by which this constraint is enforced), while the assertion

$$dubious \leftarrow emp(x),\ age(x,y),\ y > retirement\_age$$

expresses that overage employment qualifies as dubious (since, though not forbidden, it may contradict an employer's general policy that a person beyond retirement age would remain employed). Another example is the integrity constraint

$$violated \leftarrow email(x),\ sent(x,y),\ received(x,z),\ y > z$$

which declares that integrity is violated if the sent-date of an email item is after its received-date (assuming that both x and y are normalised wrt the same time zone). By contrast, the formula

$$suspect(x) \leftarrow email(x, from(y)),\ \sim authenticated(y)$$

rates an email item x received from y as suspect if the latter cannot be authenticated, although the message content of x may well be valid and unproblematic.

Although the same syntax can be used to represent conditions for integrity and quality, the use of known integrity checking methods for checking the quality of data must be deemed problematic, if not unfeasible, for the following reason.

All methods for efficient integrity checking insist that integrity must be satisfied before a given update is checked for integrity. That way, the evaluation of constraints can focus on the relevant part of the data that are actually affected by the update, while the rest can be ignored, since it is known to satisfy integrity. It would be unrealistic, however, to generalise that insistence on total integrity satisfaction by requiring that, before each update, all stored data should comply perfectly with all quality requirements. After all, certain defects of quality can never be excluded with complete certainty. Thus, not all data can be assumed to be of pristine quality whenever an update needs to be checked for quality preservation. Examples of databases where quality is not perfect but most information is useful are given by each large thesaurus or encyclopedia (think, e.g., of Wiktionary or Wikipedia).

In principle, a way out of this dilemma could be to use a method that does not insists on total integrity before each update. The only method in the literature that does not require the total satisfaction of all constraints is the so-called *brute-force* method. It exhaustively evaluates all constraints upon each update, without any simplification. But brute-force evaluation may be prohibilively expensive, due to the high complexity of constraints. Another way out could be to repair all violated constraints before or after each update.

A more elegant and less expensive solution of using integrity checking methods for monitoring quality assertions is presented in the following section.

2.3 Reconciliation

Inconsistency-tolerant integrity checking has been introduced in (Decker and Martinenghi, 2006; Decker and Martinenghi, 2008). In particular, it has been shown that, contrary to common belief, many well-known integrity checking methods, although not all of them, can waive the requirement that each consistency assertion be totally satisfied before updates can be checked efficiently. An important feature of inconsistency-tolerant methods is that none of their functionality and efficiency is compromised by arbitrarily high amounts of extant constraint violations.

Since quality properties can be expressed by the same syntax as constraints, it follows that integrity checking methods can be used to check quality properties. In particluar, the use of inconsistency-tolerant methods enables an efficient way of evaluating such properties even if there are data that do not fully comply with all quality constraints.

Nevertheless, inconsistency-tolerant methods are capable of detecting and rejecting each impairment of quality assertions upon each update, no matter if the extant imperfections are minor shortcomings or major corruptions of data. Thus, the task of improving the quality of damaged data can be delegated to separate, possibly off-line processes. Such processes may be run at any convenient point of time. In particular, they need not be run at update time, as required by traditional integrity checking approaches.

3 INCONSISTENCY TOLERANCE

In this section, we recap the main definitions of inconsistency-tolerant integrity checking (Decker and Martinenghi, 2006; Decker, 2008). Unless specified otherwise, we use terminology and notations that are conventional in the databases community (see, e.g., (Ramakrishnan and Gehrke, 2003)).

Throughout, let 'method' always signify an integrity checking method. We assume that each constraint is represented in *prenex form*, i.e., an implicit or explicit quantifier precedes a quantifier-free matrix. This includes the two most common forms of representing a constraint, either as a denial (i.e., a clause without head whose body is a conjunction of literals) or in *prenex normal form* (i.e., quantifiers outermost, negations innermost). An *integrity theory* is a set of constraints. An *update* is a bipartite finite set of database clauses to be inserted or deleted.

From now on, let the symbols D, IC, U, I and $\mathcal{M}$ always denote a database, an integrity theory, an update, a constraint and, resp., a method. We write D^U to denote the updated database, and also refer to D and D^U as the *old* and the *new* state, respectively.

We assume that the semantics of D and IC is given by a distinguished unique Herbrand model of D. Thus, I is *satisfied* (*violated*) in D if I is *true* (resp., *false*) in that model. As usual, IC is called *satisfied* (*violated*) in D if each $I \in IC$ (resp., at least one $I \in IC$) is satisfied (resp., violated) in D. For convenience, we write $D(IC) = true$ and $D(I) = true$ for denoting that IC or, resp., I is satisfied in D, and $D(IC) = false$ ($D(I) = false$) that it is violated. 'Consistency' and 'inconsistency' are synonymous with 'satisfied' and, resp., 'violated' integrity.

Each correct method $\mathcal{M}$ can be formalized as a mapping that takes as input a triple (D,IC,U) such that $D(IC) = true$, and outputs upon termination either *ok* or *ko*. Here, *ok* means that $\mathcal{M}$ accepts U because U does not violate any constraint, and *ko* means that $\mathcal{M}$ does not accept U. For inconsistency-tolerant methods, the premise $D(IC) = true$ can be waived without penalty. For simplicity, we only consider input triples (D, IC, U) such that the computation of $\mathcal{M}$ (D,IC,U) terminates. In practice, that can always be achieved by a timeout mechanism with output *ko*.

Each constraint I can be conceived as a set of particular instances, called 'cases', of I, such that I is satisfied iff all of its cases are satisfied. Thus, integrity maintenance can focus on satisfied cases, and check if their satisfaction is preserved across updates. Violated cases can thus be tolerated and possibly dealt with at any moment that is more convenient. That is captured by the following definition.

Definition. *Inconsistency-tolerant Integrity.*

a) A variable x is called a *global variable* in I if x is $\forall$-quantified in I and $\exists$ does not occur left of the quantifier of x.

b) For a constraint I and a substitution ζ of its global variables, let $I\zeta$ be obtained by replacing each global variable in I by the term assigned to it in ζ. Each such $I\zeta$ is called a *case* of I.

c) Let $\mathsf{SC}(D, IC)$ denote the set of all cases C of all $I \in IC$ such that $D(C) = true$, i.e., C is satisfied in D.

d) $\mathcal{M}$ is called *inconsistency-tolerant* if, for each triple (D, IC, U), the output $\mathcal{M}(D, IC, U) = ok$ entails that $D^U(C) = true$, for each $C \in \mathsf{SC}(D, IC)$.

In words, the definition above means: If an inconsistency-tolerant $\mathcal{M}$ accepts an update without insisting that each constraint be satisfied before the update, then the output *ok* guarantees that each case of IC that was satisfied in D remains satisfied in D^U.

Example. For relations p, q, let the second column of q be subject to the foreign key constraint $I = \forall x,y\, \exists z(q(x,y) \rightarrow p(y,z))$, which references the primary key column of p, constrained by $I' = \leftarrow p(x,y), p(x,z), y \neq z$. The global variables of I are x and y; all variables of I' are global. For U = *insert* $q(a,b)$, a typical method $\mathcal{M}$ only evaluates the simplified basic case $\exists z\, p(b,z)$ of I. If, for instance, (b,b) and (b,c) are rows in p, $\mathcal{M}$ outputs *ok*, ignoring all irrelevant violated cases such as, e.g., $\leftarrow p(b,b)$, $p(b,c)$, $b \neq c$ and I', i.e., all extant violations of the primary key constraint. $\mathcal{M}$ is inconsistency-tolerant if it always ignores irrelevant violations. $\mathcal{M}$ outputs *ko* if there is no tuple matching (b,z) in p.

It is easy to see that inconsistency-tolerant integrity checking significantly generalizes the traditional approach, which does not legitimize the use of methods in the presence of extant constraint violations.

As shown in (Decker and Martinenghi, 2006, 2008), many known methods for integrity checking are inconsistency-tolerant. The reasoning of inconsistency-tolerant methods, and also of methods that are non inconsistency-tolerant, is featured at length in subsection 4.2.

4 RISK MANAGEMENT

In this section, we illustrate how to use the evaluation of assertions by integrity checking methods for monitoring and managing risks.

4.1 Risks and Quality

A risk is a negative quality. Its positive counterpart may be characterized by properties such as security, dependability, reliability, safety and the like. Risks can often not be totally excluded, while it is always

requisite to minimize and to control them for lowering their probability to increase.

Of course, the amount of tolerable risk depends on the application and often also on its users (think, e.g., of stock market transactions). The example elaborated in 4.2 is open to interpretation. By assigning convenient meanings to predicates, it could be interpreted as a risk model of, e.g., financial services (think, e.g., of Basel II), or a nuclear power plant.

4.2 An Extended Example

Of course, a single example can always be criticized to be statistically irrelevant. However, for each of the mentioned alternatives, several typical features that are independent of the particular example are illustrated. In particular, we are going to see that, for safety-critical applications, the use of a method that is inconsistency-tolerant is more dependable than to use one which is not. Our example will show that using a non-inconsistency-tolerant method for monitoring risks may have fatal consequences.

We are going to compare inconsistency-tolerant integrity checking with the following alternative approaches to monitor risk: brute-force evaluation, non-inconsistency-tolerant integrity checking, repairing, and consistent query answering (Arenas et al., 1999). In detail, we address the following points 1) - 6).

1) The cost of the brute-force method.

2) The cost of inconsistency-tolerant methods.

3) The dependability of methods.

4) The cost of repairing the old state.

5) The cost of repairing the new state.

6) The risk of consistency query answering.

Let us consider a database D whith the following definitions of view predicates *rl*, *rm*, *rh* that model risks of low, medium and, respectively, high degree.

$$rl(x) \leftarrow p(x,x)$$
$$rm(y) \leftarrow q(x,y),\ \sim p(y,x)$$
$$rm(y) \leftarrow p(x,y),\ q(y,z),\ \sim p(y,z),\ \sim q(z,x)$$
$$rh(z) \leftarrow p(0,y),\ q(y,z),\ z > th$$

In the clause defining *rh*, let *th* be a evaluable threshold value that we assume to be always greater or equal 0. Now, let the risks be denied as in the following integrity theory:

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

Before populating D with facts about p and q, let us verify that IC is satisfiable at all by any etension of D. Indeed, it is, e.g., by each extension of p such that no fact of the form $p(0,y)$ is in p and any of the following alternatives holds: either $p = q$, or D contains $\{q(2,1), p(1,2), p(2,1)\}$ and arbitrarily many facts of the form $p(n,n+m)$, for $n > 1$, $m > 0$.

Now, let the extensions of p and q be as follows.

$$p(0,0),\ p(0,1),\ p(0,2),\ p(0,3),\ldots,p(0,10000),$$
$$p(1,2), p(2,4), p(3,6), p(4,8),\ldots,p(5000,10000)$$
$$q(0,0),\ q(1,0),\ q(3,0),\ q(5,0),\ q(7,0),\ldots,q(9999,0)$$

Clearly, there is a single violated low-risk case in D, which is caused by $p(0,0)$. Let us make sure that there is no other violated risk case in D, but trying to refute each denial about *rl*, *rm* and *rh*.

First of all, there obviously is no other low-risk cause of form $p(x,x)$ that would violate $\leftarrow rl(x)$.

Next, let us try to find an instance of the body of the first clause of *rm* that would be *true* in D. Since the second column of q is always 0, $q(x,0), \sim p(0,x)$, would have to be *true*. That, however, cannot be, since $p(0,x) \notin D$ for each x such that $q(x,0) \in D$.

For trying to find a satsified instance of the body of the second clause of *rm*, let e stand for an even number greater or equal 0, o for an odd number greater or equal 1, and n for any natural number greater or equal 0. Further note that each p-fact in D is either of the form $p(0,e)$ or $p(0,o)$ or $p(n,2n)$, for $n > 1$. So, since the second column of p joins with the first column of q only if their value is an even number, the only possible instances of that clause which could make its body *true* are of one of the following three forms:

$$p(0,e),\ q(e,z),\ \sim p(e,z),\ \sim q(z,0)$$
or
$$p(0,o),\ q(o,0),\ \sim p(o,0),\ \sim q(0,0)$$
or
$$p(n,2n),\ q(2n,0),\ \sim p(2n,0),\ \sim q(0,n)$$

Obviously, none of these instances can become *true*, because $q(e,z)$ does not hold for any z, $q(0,0)$ is *true* in D, and $q(2n,0)$ is *false* for each $n > 0$.

Last, the clause of *rh*: to make its body *true* would require that $0 > th$, but we have excluded that. Hence, we have verified that $\leftarrow rl(0)$ is the only violated risk case of IC in D, and that $p(0,0)$ is its only cause.

Now, consider U = *insert* $q(0,9999)$, for illustrating 1) - 6) above.

1) The cost of brute-force checking for any update is high. That is a commonplace, but let us see in some more detail to what brute-force evaluation of IC amounts, for later comparison.

Evaluation of $\leftarrow rl(x)$ involves a scan of all of p. Evaluation of $\leftarrow rm(x)$ involves joins of p and q, a join of local p with remote q, plus possibly many lookups in p and q. Evaluation of $\leftarrow rh(x)$ involves a join of local p with remote q, plus the evaluation of

possibly many ground expressions of the form $z > th$.

With large extensions of p and q, the evaluation steps outlined above may last too long, particularly if safety-critical risks are monitored in real time. In the following point, we shall see that it is far less expensive to use an inconsitency-tolerant method that simplifies the evaluation of integrity constraints by taking the update into account and by limiting its focus on the data that are affected by the update.

2) We are going to see that the cost of inconsistency-tolerant integrity checking of U is much lower than to use brute-force evaluation. But, before we go into details, recall that the use of any traditional method that insists on the satisfaction of *IC* in the old state D is prohibited for the database in our example, since $D(IC) = false$.

Typical simplification methods compile pre-simplifications for update patterns at constraint specification time. Thus, the cost of such pre-simplifications at update time is nil. U matches the update pattern $q(a,b)$, which in turn matches precisely the following unfoldings of $\leftarrow rm$ by the two clauses defining *rm*, and of $\leftarrow rh$, respectively.

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

Thus, the pre-simplifications complied for the patter for insertions of facts of the form $q(a,b)$, are as follows.

$$\leftarrow \sim p(b,a)$$
$$\leftarrow p(x,a),\ \sim p(a,b),\ \sim q(b,x)$$
$$\leftarrow p(0,a),\ b > th$$

Substituting (a,b) by the inserted values $(0,9999)$ at update time yields the following simplifications.

$$\leftarrow \sim p(9999,0)$$
$$\leftarrow p(x,0),\ \sim p(0,9999),\ \sim q(9999,x)$$
$$\leftarrow p(0,0),\ 9999 > th$$

By a simple lookup of $p(9999,0)$ for evaluating the first of the three denials, it is inferred that $\leftarrow rm$ is violated.

Since a medium risk has been detected, there is in principle no need to continue checking the remaining two simplified denials. However, we are going to do that, in order to build a bridge to point 3).

Evaluating the second denial from left to right amounts to the cost of answering the query $\leftarrow p(x,0)$. The single answer is $x = 0$. Then, a lookup of $q(9999,0)$ succeeds. Hence, the second denial is *true*, which means that there is no further medium risk.

Since $p(0,0)$ is *true*, the third denial turns out to be violated if $9999 > th$ holds, thus indicating a high security risk.

To summarize this point: Inconsistency-tolerant integrity checking of U essentially costs a simple access to the p relation. Moreover, if all constraints are evaluated even after some violation has been detected, only an additional simple lookup is needed. And, perhaps more importantly, inconsistency-tolerant integrity checking prevents medium- and high-risk violations that would be caused by the update if it were not rejected.

3) Inconsistency-tolerant checking is dependable, non-inconsistency-tolerant checking is not. This claim is confirmed by considering the following kind of reasoning, as performed by methods that are not inconsistency-tolerant. Such methods are specified, e.g., in (Gupta et al., 1994; Lee and Ling, 1996).

Since the p relation is not affected by U, the truth value of the unfolding $\leftarrow p(x,x)$ of the constraint $\leftarrow rl(x)$ is the same in D. Since each method that is not inconsistency-tolerant insists on the premise that all constraints be satisfied in the old state, such methods, when applied to our example, conclude that the unfolded denial $\leftarrow p(x,x)$ is *true* in D and D^U, even though $p(0,0) \in D$. That conclusion is then applied to the third of the simplified unfoldings from 2), which is reprinted below, for convenience.

$$\leftarrow p(0,0),\ 9999 > th$$

The subsumption-based reasoning of methods that are not inconsistency-tolerant can be summarized as follows: Applying the premise that $\leftarrow p(x,x)$ is satisfied to $\leftarrow p(0,0), 9999 > th$ infers that the latter also remains satisfied in D^U, because it is subsumed by $\leftarrow p(x,x)$. Thus, non-inconsistency-tolerant integrity checking wrongly concludes that the high risk constraint $\leftarrow$ *rh(z)* is not violated in D^U.

4) Now, we are going to see that repairing the old state is costly. Recall that the traditional integrity checking approach insists on total constraint satisfaction in the old state. This means that all extant violations need to be repaired before each update. In general, the identification of all extant constraint violations may already be very expensive in large databases, and indeed unaffordable at update time.

Fortunately, however, there is only a single low-risk constraint violation in our example, as we have already seen before: $p(0,0)$ is the only cause of the only constraint violation $\leftarrow$ *rl(0)* in D. Thus, to repair D means to first delete $p(0,0)$, and then check if that preserves all assertions.

To check *delete* $p(0,0)$ for integrity means to

check the simplified denials

$$\leftarrow q(0,0)$$

and

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

obtained from resolving ${\sim}p(0,0)$ with the bodies of the two clauses defining *rm*, since precisely those two clauses are affected by the deletion of $p(0,0)$. Hence, no constraint other than $\leftarrow rm(y)$ is potentially violated by the intended repair.

Of the two simplified denials above, the second one clearly is satisfied in D^U, since no fact of the form $p(x,0)$ remains in the database after $p(0,0)$ is deleted. However, the first one is violated, since $q(0,0)$ is *true* in D^U. Hence, another repair action is needed. The obvious candidate is *delete* $q(0,0)$.

To delete $q(0,0)$ affects

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

and yields the simplified check of

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

Obviously, this denial is violated by all facts in D that are of the form $p(0,o)$ and $q(o,0)$, where o is an odd number in the interval $[1,9999]$. Thus, to delete $q(0,0)$ for repairing the violation caused by deleting $p(0,0)$ causes the violation of each case of the form $\leftarrow rm(o)$, for each odd number o in $[1,9999]$.

Clearly, many facts about p or q would have to be deleted in order to repair each of these violated cases. For simplicity, we won't follow them through, since the point that repairing D is very complex and tends to be much more expensive than inconsistency-tolerant integrity checking has become obvious already. We only recall the big advantage of inconsistency-tolerant integrity checking that repair actions do not have to take place at update time. Instead, they can be taken off-line, at any convenient moment.

5) Also repairing the new state is more costly than to simply tolerate extant constraint violations until they can be repaired at some better moment. In our example, this becomes obvious by recalling from 1) that, in D^U, there are three violated cases: the low-risk case that is already violated in D and the medium- and high risk cases as detected by inconsistency-tolerant integrity checking. To repair them is indeed even more complicated than to only repair the violated low-risk case, as attempted in 4).

Moreover, it should be noted for risk management that it is no good idea in general to simply accept an update without checking for potential violations of constraints, and to attempt repairs only after the update is committed, because repairing takes time, during which an updated but unchecked state may contain malicious risks of any order.

6) Consistent query answering in inconsistent databases (CQA) is a popular approach to cope with extant constraint violations for query answering (Arenas et al., 1999). Although query answering is not the topic of this paper, a connection between inconsistency-tolerant integrity checking and CQA can easily be drawn, because the monitoring of quality constraints involves the evaluation of such constraints or their simplifications. Thus, the idea may arise to use CQA for evaluating constraints as queries, in order to avoid wrong answers hat could be due to extant constraint violations.

However, to evaluate constraints or simplifications thereof by CQA is not recommendable, because consistent answers are defined to be those that are true in each minimally repaired state of the datbase. Thus, for each queried constraint, CQA will by definition return the empty answer, which indicates the satisfaction of the constraint. Thus, answers to queried constraints that are computed by CQA have in fact no meaningful interpretation.

For instance, CAQ computes the empty answer to the query $\leftarrow rl(x)$ as well as to the query $\leftarrow rh(z)$, for any extension of the relations p and q. However, the only sensibly correct answer to the first query in D is $x = 0$. Similarly, the only reasonable answer to the second query in D^U is $x = 9999$, assuming that $9999 > th$. These answers are reasonable because they correctly indicate risks contained in D and D^U, respectively.

This shows that, despite of many unquestionable merits of CQA, it should not be used for monitoring quality, if quality is modeled by integrity constraints.

5 RELATED WORK

Although database quality and database integrity are intuitively related, they never have been approached in a uniform manner, to the best of our knowledge, neither in theory nor in practice.

Kinships and semantic differences between data that have or lack quality, and data that have or violate integrity are observed, in a collection of work on modeling and managing uncertain data (Motro and Smets, 1996). In that book, largely diverse approaches to handle data that lack quality are proposed. In particular, approaches such as probabilistic and fuzzy set modeling, exception handling, repairing and paraconsistent reasoning are discussed. However, no particular approach to integrity checking is considered.

Integity checking also has never been addressed in detail by related work on consistent query answering in inconsistent databases (Arenas et al., 1999).

Moreover, several paraconsistent logics that tolerate inconsistency and quality impairment of data have been proposed, e.g., in (Decker et al., 2002; Bertossi et al., 2005). Each of them, however, departs from classical first-order logic, by adopting some annotated, probabilistic, modal or multivalued logic, or by eliminating and replacing standard axioms and inference rules with non-standard axiomatizations. As opposed to that, inconsistency-tolerant integrity checking fully conforms with standard two-valued datalog and does not need any extension of classical logic.

Further work on the management of inconsistencies in databases is going on in the field of measuring inconsistency (Grant and Hunter, 2006; Decker, 2009). Inconsistency measures can be used for a form of inconsistency-tolerant integrity checking that is different from the approach outlined in section 3. It accepts an update only if the measured amount of inconsistency in the old state does not increase in the new state. There are several possible ways to measure inconsistency. Two that are directly related to section 3 are to count the number of violated cases, or to compare the set of violated cases before and after the update (Decker and Martinenghi, 2008). In (Decker, 2009), some more measures related to inconsistency-tolerant integrity checking are discussed.

6 CONCLUSIONS

We have shown that the quality and the integrity of stored data can be modeled and monitored in a straightforward, uniform manner. Conditions that capture properties of data quality and integrity can both be modeled by database assertions.

Traditionally, no method for simplified integrity checking has tolerated as input a database that is inconsistent with its constraints. However, as shown in (Decker and Martinenghi, 2006, 2008), it is possible to waive that restriction. Hence, quality and integrity assertions, the occasional violation of which is tolerable, can be monitored efficiently by inconsistency-tolerant constraint checking methods.

It is important to note that, for achieving inconsistency tolerance, no re-implementation or extension of any existing method that can be shown to have that property is needed. As illustrated in the extended example of this paper, inconsistency tolerance is essential, since wrong, possibly fatal conclusions can be inferred from deficient data by using a method that is not inconsistency-tolerant. Many methods that are inconsistency-tolerant, and also some that are not, have been identified in (Decker and Martinenghi, 2006, 2008).

Ongoing work is concerned with establishing a closer relationship of inconsistency-tolerant integrity checking with the fields of repairing, consistent query answering and inconsistency measuring.

REFERENCES

Arenas, M., Bertossi, L. E., and Chomicki, J. (1999). Consistent query answers in inconsistent databases. In *Proceedings of PODS*, pages 68–79. ACM Press.

Bertossi, L., Hunter, A., and Schaub, T. (2005). *Inconsistency Tolerance*, volume 3300 of *LNCS*. Springer.

Bowen, K. and Kowalski, R. A. (1982). Amalgamating language and metalanguage. In Clark, K. and Tärnlund, S.-A., editors, *Logic Programming*, pages 153–172. Academic Press.

Decker, H. (2008). Inconsistency-tolerant integrity checking for knowledge assimilation. In Filipe, J., Shishkov, B., Helfert, M., and Maciaszek, L., editors, *Software and Data Technologies*, volume 22 of *CCIS*, pages 320–331. Springer.

Decker, H. (2009). Quantifying the quality of stored data by measuring their integrity. Submitted.

Decker, H. and Martinenghi, D. (2006). A relaxed approach to integrity and inconsistency in databases. In Hermann, M. and Voronkov, A., editors, *Proc. 13th LPAR*, volume 4246 of *LNCS*, pages 287–301. Springer.

Decker, H. and Martinenghi, D. (2008). Classifying integrity checking methods with regard to inconsistency tolerance. In *Proceedings of the 10th ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, pages 195–204. ACM Press.

Decker, H., Villadsen, J., and Waragai, T., editors (2002). *Proceedings of the ICLP 2002 workshop on Paraconsistent Computational Logic*, volume 95 of *Datalogiske Skrifter*. Roskilde University, Denmark.

Grant, J. and Hunter, A. (2006). Measuring inconsistency in knowledgebases. *Journal of Intelligent Information Systems*, 27(2):159–184.

Gupta, A., Sagiv, Y., Ullman, J. D., and Widom, J. (1994). Constraint checking with partial information. In *Proceedings of PODS 1994*, pages 45–55. ACM Press.

Lee, S. Y. and Ling, T. W. (1996). Further improvements on integrity constraint checking for stratifiable deductive databases. In *VLDB'96*, pages 495–505. Kaufmann.

Motro, A. and Smets, P. (1996). *Uncertainty Management in Information Systems: From Needs to Solutions*. Kluwer.

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill.