Inconsistency-tolerant Integrity Checking

Hendrik Decker and Davide Martinenghi

***Abstract*— All methods for efficient integrity checking require all integrity constraints to be totally satisfied, before any update is executed. However, a certain amount of inconsistency is the rule, rather than the exception in databases. In this paper, we close the gap between theory and practice of integrity checking, i.e., between the unrealistic theoretical requirement of total integrity and the practical need for inconsistency tolerance, which we define for integrity checking methods. We show that most of them can still be used to check whether updates preserve integrity, even if the current state is inconsistent. Inconsistency-tolerant integrity checking proves beneficial both for integrity preservation and query answering. Also, we show that it is useful for view updating, repairs, schema evolution and other applications.**

***Index Terms*— Integrity Checking, Inconsistency Tolerance.**

I. Introduction

Integrity constraints are statements declared in the database schema. They express semantic properties, meant to be invariably satisfied by the stored data across state changes.

For preserving the satisfaction of simple constraints like primary keys, foreign keys, or CHECK constraints, sufficient support is usually provided by the database management system (DBMS). For constraints that are not supported by the DBMS, the majority of scientific publications on the subject proposes to use some automated, application-independent method for integrity checking.

Each such method takes as input the set of constraints in the schema, an update consisting of two (possibly empty) sets of database elements to be inserted or, respectively, deleted, and possibly the current, also called 'old' state of the database. The output of the methods indicates whether the 'new' state, obtained from updating the old state, would satisfy or violate integrity.

In theory, each method requires the *total integrity* of the old state, i.e., no violation whatsoever is tolerated at any time. Total integrity, however, is the exception, rather than the rule in practice.

Integrity violation may sneak into a database in many ways. For instance, new constraints may be added without being checked for violations by legacy data. Or, integrity control may be turned off temporarily, e.g., when uploading a backup for which a total check would last too long. Or, integrity may deteriorate by migrating to the DBMS of a different vendor, since the semantics of integrity constructs tends to be proprietary. Or, integrity may be compromised by the integration of databases, when constraints that had held locally fail to hold after databases have been merged.

Other database applications where inconsistencies may occur are view updating, schema evolution, data mining and warehousing, diagnosis, replication, uncertain data, and many more.

Often, users consider efforts to completely repair all inconsistencies unnecessary, inopportune, unaffordable or impossible. Violations of constraints may even be desirable, e.g., when constraints are used to detect irregularities, such as indications of security attacks, tax dodging, etc. So, even though the standard logic foundations are intolerant wrt. inconsistency, there is a strong practical need for integrity checking methods that are able to tolerate extant cases of constraint violations.

For convenience, we abbreviate, from now on, inconsistency-tolerant integrity checking by *ITIC*.

Fortunately, no new methods for ITIC have to be invented. The main purpose of this paper is to show that the gap between theory and practice of integrity checking can be closed by already approved, time-tested methods. Contrary to common belief, these methods can waive the unrealistic requirement of total integrity satisfaction, without forfeiting their capacity to check integrity, even in the presence of inconsistency. Our approach to ITIC yields major quality improvements, both of data wrt. their intended semantics, and of answers to queries.

The aims pursued in this paper are the following.

1) *To distinguish methods that are inconsistency-tolerant from those that are not.* In this paper, we formalize the notion of ITIC. Before that, the behavior of methods for checking declaratively stated constraints in the presence of inconsistency has never been contemplated. Traditionally, integrity checking methods were not legitimized to be used in the presence of inconsistency, although many databases are not totally consistent. Now, inconsistency-tolerant methods can be soundly used in the presence of an arbitrary amount of inconsistency. Thus, the applicability of integrity checking methods is widened immensely. To the best of our knowledge, our definition is the first of its kind.

2) *To bridge the gap between theory and practice of integrity checking by using inconsistency tolerance.* The theoretical total-integrity requirement is a formidable desideratum in practice. Typically, practical approaches to deal with extant inconsistency are based on exception handling. They tend to have the character of workarounds or ad-hoc solutions. Theoretical approaches to deal with extant inconsistency have been based on non-classical logics such as modal, many-valued or paraconsistent calculi. Our approach is based on classical logic and does not need any changes or adaptations of existing integrity checking methods.

3) *To evaluate the effects of ITIC on database evolution and query answering.* Ultimately, integrity checking is about preserving the semantics of data through updates and, consequently, obtaining query answers that can be trusted. Without total integrity, full trustability is lost. Yet, some databases may be less inconsistent than others, and thus better behaved wrt. query answering. In this paper, we propose a comprehensive set of experiments for observing the impact of ITIC on databases subject to evolution through updates. We report both on the number of constraint violations and on the number of incorrect answers to complex benchmark queries. We also compare our approach to consistent query answering [1], which is an orthogonal technique for dealing with inconsistent data.

4) *To describe several application contexts that may benefit from ITIC*. The vision brought forward in this paper can be applied

H. Decker is with the Instituto Tecnológico de Informática, Campus de Vera, edificio 8G, E-46071 Valencia, Spain, (hendrik@iti.upv.es).

D. Martinenghi is with Politecnico di Milano, Dipartimento di Elettronica e Informazione, piazza Leonardo da Vinci 32, I-20133 Milano, Italy, (martinen@elet.polimi.it).

Digital Object Indentifier 10.1109/TKDE.2010.87

to various knowledge and data management problems. We show that ITIC naturally extends to view updates, database repairs, schema evolution, risk management, and unsatisfiability handling.

Section II outlines the background. The main contributions are: to develop a concept of ITIC (Section III), to show the inconsistency tolerance of known methods (Section IV), to outline several applications of ITIC (Section V), and to validate the practical relevance of ITIC (Section VI). Related work is discussed in Section VII. In Section VIII, we conclude.

II. Preliminaries

We adopt the usual terminology and notation of *datalog*, and refer to textbooks in the field (e.g., [2]) for further background.

A. Logic and Databases

Throughout, let symbols a, b, ... denote *constants*, p, q, ... *predicates* and x, y, ... *variables*. A *term* is either a variable or a constant. Sequences of terms are denoted as vectors, e.g., $\vec{t}$. Predicates, terms, logical connectives $\sim, \wedge, \vee, \leftarrow$, 0-ary predicates *true*, *false*, and quantifiers $\forall$, $\exists$ are used in *formulas*, defined as follows: *i)* if p is an n-ary predicate and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is a formula; *ii)* if F and G are formulas then so are $\sim F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$; *iii)* if F is a formula and x a variable such that neither $\forall x$ nor $\exists x$ occurs in F, then $\forall x F$ and $\exists x F$ are formulas; in $\forall x F$ and $\exists x F$, each occurrence of x in F is said to be *bound*. A formula in which all variables are bound is said to be *closed*. A formula preceded by $\sim$ is said to be *negated*. Formulas of the form $p(t_1, \ldots, t_n)$, where p is a predicate and the t_i are terms, are called *atoms*. A *literal* is either an atom (*positive literal*) or a negated atom (*negative literal*).

An *expression* is either a formula or a term. A *substitution* σ is a set of pairs of terms $\{x_1/t_1, \ldots, x_n/t_n\}$, where $x_1, \ldots, x_n$ are distinct variables; let $\{x_1, \ldots, x_n\}$ be denoted by *Dom*(σ). The *restriction* of a substitution σ to a set of variables $\mathcal{V} \subseteq$ *Dom*(σ) is the substitution $\sigma' \subseteq \sigma$ such that *Dom*$(\sigma') = \mathcal{V}$.

For an expression E (or a set of expressions $\mathcal{E}$) and a substitution σ, the expression $E\sigma$ ($\mathcal{E}\sigma$) be obtained by replacing each occurrence of each variable from *Dom*(σ) in E ($\mathcal{E}$) by the corresponding term in σ. An expression is called *ground* if it contains no variable. A substitution σ is *more general* than a substitution θ if there is a substitution ϕ such that, for each expression E, $E\theta = (E\sigma)\phi$. A substitution σ is a *unifier* of expressions $E_1, \ldots, E_n$ if $E_1\sigma = \cdots = E_n\sigma$; σ is a *most general unifier* (*mgu*) of $E_1, \ldots, E_n$ if σ is more general than any other unifier of $E_1, \ldots, E_n$.

A *clause* is a formula of the form $H \leftarrow B_1 \wedge \cdots \wedge B_n$ ($n \geq 0$), where H is a positive literal, $B_1, \ldots, B_n$ are literals and, implicitly, each variable in $H \leftarrow B_1 \wedge \cdots \wedge B_n$ is universally quantified in front of the clause. H is called the *head* and $B_1 \wedge \cdots \wedge B_n$ the *body* of the clause. The head is optional; when absent, the clause is called a *denial*, and its body can be read as a condition that must not hold. The *empty clause* is a denial with an empty body; it is equivalent to *false*. A *fact* is a clause whose head is ground and whose body is empty.

A *database clause* is a clause with non-empty head $H \notin \{$*true*,*false*$\}$. A *database* is a finite set of database clauses. The dependency graph $\mathcal{D}_D$ of a database D is a directed graph such that its nodes are labeled with the predicates in D, and there is a positive (resp., negative) arc (p, q) in $\mathcal{D}_D$ for each clause $H \leftarrow B$ in D and each pair of predicates p, q such that q occurs in H and p in a positive (resp., negative) literal in B. A database D is *relational* if each clause in D is a fact; D is *hierarchical* if no cycle exists in $\mathcal{D}_D$, i.e., no predicate recurs on itself; D is *stratified* if no cycle with a negative arc exists in $\mathcal{D}_D$, i.e., no predicate recurs on its own negation.

An *update* is a bipartite finite set of clauses to be deleted and inserted, respectively. For a database D and an update U, let D^U denote the updated database; we also call D and D^U the *old* and the *new state*, respectively. For a fact A in U to be inserted or deleted, we may write "*insert* A" or, resp., "*delete* A".

B. Integrity

We are going to formalize basic notions of database integrity.

1) Syntax: An *integrity constraint* (in short *constraint*) is a closed first-order predicate logic formula. As usual, constraints are represented either a denials or in *prenex conjunctive normal form* (*PCNF*), i.e., formulas of the form $I = QI'$, where Q is a sequence of quantified variables $Q_1 x_1 \ldots Q_n x_n$, each Q_i is either $\forall$ or $\exists$, and the so-called *matrix* I' is a conjunction of disjunctions of literals.

A variable x in a constraint I is called a *global variable* in I if x is $\forall$-quantified and $\exists$ does not occur left of $\forall x$ in the PCNF of I. Let *Glb*(I) denote the set of global variables in I.

An *integrity theory* is a finite set of integrity constraints.

2) Semantics: We use *true* and *false* also to denote truth values. We only consider databases that have a two-valued semantics, given by a unique standard model, e.g., stratified databases with the stable model semantics [3]. That also determines the semantics of integrity, as follows.

Let I be a constraint, *IC* an integrity theory, and D a database. We write $D(I) =$ *true* (resp., $D($*IC*$) =$ *true*) and say that I (resp., *IC*) is *satisfied* in D if I (resp., each constraint in IC) is *true* in D. Else, we write $D(I) =$ *false* (resp., $D($*IC*$) =$ *false*) and say that I (resp., *IC*) is *violated* in D.

In the literature, the semantics of integrity is not always defined by the truth or falsity of constraints, as in the preceding definition. For instance, the "consistency view" in [4] defines satisfaction not by truth, but by satisfiability. The "theoremhood view" in [5] defines that a constraint is violated if it is not *true*, which does not necessarily mean that it is *false*, e.g., in the completion of databases with predicates defined by recurring on themselves. The preceding definition avoids such incongruences, as long as only databases with a unique two-valued model are considered.

3) Soundness and Completeness: Each integrity checking method $\mathcal{M}$ can be formalized as a function that takes as input a database, an integrity theory and an update, and outputs either *sat* or *vio*. To compute this function usually is much more efficient than the *brute-force* method, henceforth denoted by $\mathcal{M}_{bf}$, which exhaustively evaluates all constraints upon each update.

The soundness and completeness of integrity checking methods can now be generically defined as follows.

***Definition** 2.1:* [Sound and complete integrity checking] *An integrity checking method $\mathcal{M}$ is called* sound *or, resp.,* complete*, if, for each database D, each integrity theory IC such that $D($IC$) =$ true, and each update U,* (1) *or, resp.,* (2) *holds.*

$$\textit{If } \mathcal{M}(D, IC, U) = \textit{sat} \textit{ then } D^U(IC) = \textit{true}. \quad (1)$$

$$\textit{If } D^U(IC) = \textit{true} \textit{ then } \mathcal{M}(D, IC, U) = \textit{sat}. \quad (2)$$

Definition 2.1 only states soundness and completeness properties for the output *sat* of $\mathcal{M}$. Symmetrically, soundness and completeness properties for the output *vio* could be defined. We refrain from doing so, since, under the additional condition that $\mathcal{M}$ terminates, it is easy to show that soundness and completeness for *sat* is equivalent to completeness and, resp., soundness for *vio*.

Soundness, completeness and termination have been shown for the methods in [6], [5], [4], [7] and others. Other methods (e.g., [8], [9]) are only shown to be sound. Thus, they provide sufficient but not necessary conditions for guaranteeing integrity.

4) Simplifications: Most methods for efficient integrity checking attemp to "simplify" the constraints that are potentially violated by an update U, so that computing $\mathcal{M}(D, \mathit{IC}, U)$ becomes more efficient than querying all constraints by brute force.

***Example** 2.1:* Let $p(\mathit{ISBN}, \mathit{TITLE})$ be a relation with predicate p about published books, and I the constraint

$$\leftarrow p(x, y) \wedge p(x, z) \wedge y \neq z\,.$$

I states that no two books with the same ISBN may have different titles. Let U be an update that inserts $p(i, t)$. For any database D, most methods $\mathcal{M}$ compute $\mathcal{M}(D, \{I\}, U)$ by evaluating $D(I')$, where I' is the simplified constraint $\leftarrow p(i, y) \wedge y \neq t$. It states that no book with ISBN i may have a title different from t. If $\mathcal{M}$ is sound (and complete), the new state D^U is guaranteed to satisfy I if (and, resp., only if) $D^U(I') = \mathit{true}$. □

Such simplifications typically yield major gains in efficiency, as can be seen by comparing I and I' in Example 2.1. For any given update pattern, simplifications can be generated even without depending on any database state, but only on the schema and the integrity theory. Thus, database performance is not affected, since simplifications can be anticipated ahead of update time. For instance, take i and t in Example 2.1 as placeholders for actual ISBNs and titles. For the insertion of a concrete fact, e.g. $p(17, \mathit{abc})$, values 17 and *abc* replace i and, resp., t in I'. The cost of checking the resulting simplification then is that of a table look-up, while the brute-force evaluation of I would be quadratic in the size of the extension of p (if no index is used).

III. Inconsistency Tolerance in Integrity Checking

The motivation behind this paper is the need for methods that are capable of checking constraints without insisting on total integrity satisfaction. No method has ever been defined without requiring total integrity, which was thought of as indispensable. However, inconsistencies often are unavoidable, or even useful (e.g., for diagnosis, or mining fraudulent data). Thus, extant cases of violated constraints should be tolerable. Nonetheless, integrity checking should prevent that any *new* cases of integrity violation are introduced. That is captured by the definitions in III-A.

A. The Main Definitions

The goal of this section is to characterize methods that can tolerant extant cases of constraint violation in databases. For attaining that goal, we first formalize what we mean by "case".

***Definition** 3.1:* [Case]
Let I be a constraint and σ a substitution. $I\sigma$ is called a case *of I if $\mathit{Dom}(\sigma) = \mathit{Glb}(I)$; it is a* basic *case if $\mathit{Glb}(I\sigma) = \emptyset$. For a database D and an integrity theory IC, let $\mathsf{S}(D, \mathit{IC})$ denote the set of all cases C of all constraints in IC such that $D(C) = \mathit{true}$.*

***Example** 3.1:* Consider two relations with predicates r, s, and the foreign key constraint $I = \forall x, y\, \exists z(s(x, y) \rightarrow r(x, z))$ on the first argument of s, which references a primary key, the first argument of r. The global variables of I are x and y. For a fact $s(a, b)$ to be inserted, integrity checking methods usually focus on the (basic) case $\exists z(s(a, b) \rightarrow r(a, z))$ of I. It requires the existence of a fact in r whose primary key value matches the foreign key value of the inserted fact. Other cases are ignored. □

Example 3.1 illustrates the crucial role of cases for ITIC. Intuitively, at most those cases whose global variables match with values of the update need to be checked. All other cases can be ignored, even if they are violated. Most methods in the literature and in practice work that way: they focus on cases that may be violated by updated facts, while ignoring extant violations. However, the traditional theory of simplified integrity checking, anchored in Definition 2.1, does not reflect that focus. Rather, it coarsely treats each constraint I as either satisfied or violated. It does not consider that, e.g., only a few, tolerable cases of I may be violated, while all others are satisfied. The following definition does.

***Definition** 3.2:* [Inconsistency-tolerant integrity checking]
An integrity checking method $\mathcal{M}$ is sound *or, resp.,* complete wrt. inconsistency tolerance *if, for each database D, each integrity theory IC, and each update U, (3) or, resp., (4) holds.*

$$\textit{If } \mathcal{M}(D, \mathit{IC}, U) = \mathit{sat} \textit{ then } \mathsf{S}(D, \mathit{IC}) \subseteq \mathsf{S}(D^U, \mathit{IC}). \quad (3)$$

$$\textit{If } \mathsf{S}(D, \mathit{IC}) \subseteq \mathsf{S}(D^U, \mathit{IC}) \textit{ then } \mathcal{M}(D, \mathit{IC}, U) = \mathit{sat}\,. \quad (4)$$

As opposed to the traditional Definition 2.1, Definition 3.2 does not require total integrity, i.e., it may well be that $D(\mathit{IC}) = \mathit{false}$. However, in both definitions, the same function $\mathcal{M}(D, \mathit{IC}, U)$ is used for integrity checking. Thus, no new method needs to be invented for achieving inconsistency tolerance. Rather, any traditional method can be employed if it complies with (3).

***Example** 3.2:* Let I be as in Example 2.1. Let D consist of $p(1, a)$ and $p(1, b)$. Clearly, $D(I) = \mathit{false}$. Let $U = \{\mathit{insert}\ p(2, c)\}$. The simplification $\leftarrow p(2, y) \wedge y \neq c$, as obtained in Example 2.1, is *true* in D^U. Each method $\mathcal{M}$ that evaluates this simplification outputs *sat*, i.e., U is accepted because it does not introduce any violation of integrity. Thus, $\mathcal{M}$ guarantees that all cases of I that were satisfied in D remain satisfied in D^U, while tolerating the inconsistency of violated cases of I. □

Several non-trivial examples and counter-examples for Definition 3.2 are featured in section IV. A trivial example of a method that is sound wrt. inconsistency tolerance is $\mathcal{M}_{bf}$. However, $\mathcal{M}_{bf}$ is not complete wrt. inconsistency tolerance, as shown below.

***Example** 3.3:* Let D, I and U be as in Example 3.2. Clearly, $\mathcal{M}_{bf}(D, \{I\}, U) = \mathit{vio}$, but the only violated basic case of I in D^U, $\leftarrow p(1, a) \wedge p(1, b) \wedge a \neq b$, was already violated in D. □

Theorem 1 below states that ITIC generalizes the traditional approach which insists on total integrity. The generalization is proper, i.e., some but not all methods are inconsistency-tolerant, as we shall see in IV-D.

***Theorem** 1:* Let $\mathcal{M}$ be a method for integrity checking. Then, for each database D, each integrity theory *IC* such that $D(\mathit{IC}) = \mathit{true}$, and each update U, the implications (3) ⇒ (1) and (4) ⇒ (2) hold.

Proof: To show (3) ⇒ (1), note that $D(\mathit{IC}) = \mathit{true}$ entails $\mathit{IC} \subseteq \mathsf{S}(D, \mathit{IC})$. Hence, if (3) holds and $\mathcal{M}(D, \mathit{IC}, U) = \mathit{sat}$, the conclusion of (3) entails $D^U(C) = \mathit{true}$ for each $C \in \mathit{IC}$. Hence (1) follows. Similarly, (4) ⇒ (2) can be shown. ■

Theorem 1 entails that relaxing traditional integrity checking (which requires total integrity) to ITIC causes no loss of efficiency

and no extra cost at all. On the other hand, the gains are immense: with an inconsistency-tolerant method, database operations can proceed even in the presence of (obvious or hidden, known or unknown) violations of integrity. As opposed to that, integrity checking was traditionally not legitimate in the presence of constraint violations, i.e., it had to wait until integrity was repaired.

Example 3.4 below illustrates another advantage of ITIC: each update that does not introduce any new case of integrity violation can be accepted, while extant violated cases may disappear, intentionally or accidentally.

***Example** 3.4:* Let D and I be as in Example 3.2, and let $U = \{\textit{delete } p(1,b)\}$. Since $D(I) = \textit{false}$, no method that insists on total integrity is in the position to check this update. However, each inconsistency-tolerant method is. Each method that is complete wrt. inconsistency tolerance (and in fact each method assessed in Section IV) returns *sat* for this example, since U does not introduce any new violation of I. Since U even repairs a violated constraint, there are several reasons to accept this update, and the lack of total integrity is no reason to reject it. □

B. Sufficient and Necessary Conditions

We are now going to discuss conditions that will be used for assessing the inconsistency tolerance of methods in section IV. Conditions (5) and (6) below are sufficient for soundness (3) and, resp., completeness (4), as shown in Theorem 2. Later, (8), which is also interesting on its own, is shown to be necessary for (4).

$$\begin{array}{l}\text{If } \mathcal{M}(D, \mathit{IC}, U) = \mathit{sat} \\ \text{then, for each } C \in \mathsf{S}(D, \mathit{IC}),\ \mathcal{M}(D, \{C\}, U) = \mathit{sat}\end{array} \tag{5}$$

$$\begin{array}{l}\text{If, for each } C \in \mathsf{S}(D, \mathit{IC}),\ \mathcal{M}(D, \{C\}, U) = \mathit{sat} \\ \text{then } \mathcal{M}(D, \mathit{IC}, U) = \mathit{sat}\end{array} \tag{6}$$

***Theorem** 2:* Let $\mathcal{M}$ be a sound method for integrity checking. Then, for each database D, each integrity theory *IC*, the implications (5) ⇒ (3) and (6) ⇒ (4) hold.

Proof: By applying (1), the "then" part of (5) becomes

$$\text{for each } C \in \mathsf{S}(D, \mathit{IC}),\ D^U(C) = \mathit{sat}, \tag{7}$$

which is the same as $\mathsf{S}(D, IC) \subseteq \mathsf{S}(D^U, IC)$, hence the thesis. Similarly, applying (1) on the "if" part of (6) yields (4). ■

In Section IV, condition (5) is verified for the methods in [6], [5], [4], and (6) is verified for [6]. Interestingly, we are going to see that many other methods turn out to not fulfill (4), since, e.g., they may output *vio* whenever an update yields a redundant new path for deriving some already violated case. So, if the update causes no other violation of integrity, the premise of (4) holds, but its conclusion does not. In other words, the output *vio* of methods that are sound but incomplete wrt. inconsistency tolerance does not guarantee that the given update would violate a case of some constraint that was satisfied in the old state. However, the following, somewhat weaker property holds for several methods.

***Definition** 3.3:* [Weakly complete inconsistency tolerance] *Let $\mathcal{M}$ be a method for integrity checking. $\mathcal{M}$ is called* weakly complete wrt. inconsistency tolerance *if, for each database D, each integrity theory IC and each update U, the following holds.*

$$\textit{If } D^U(\mathit{IC}) = \textit{true then } \mathcal{M}(D, \mathit{IC}, U) = \mathit{sat}\,. \tag{8}$$

The technical difference between (2) and (8) is that, for (2), total integrity of the old state is required, but not for (8).

In practice, weak completeness wrt. inconsistency tolerance is a desirable property: The output *vio* of any sound but incomplete integrity checking method means that further checking is needed for deciding if the update preserves or violates integrity. However, the contraposition of (8) ensures that, if a weakly complete method outputs *vio*, integrity surely is violated after the update, i.e., no further checking is needed. In fact, it is easy to show the following direct consequences of Definitions 2.1, 3.2 and 3.3.

***Corollary** 3:* Let $\mathcal{M}$ be a method for integrity checking.
a) If $\mathcal{M}$ is complete wrt. inconsistency tolerance, then it is also weakly complete wrt. inconsistency tolerance ((4) ⇒ (8)).
b) If $\mathcal{M}$ is weakly complete wrt. inconsistency tolerance, then it is also a complete integrity checking method ((8) ⇒ (2)).

TABLE I
PROPERTIES OF INTEGRITY CHECKING METHODS

	$\mathcal{M}_N$	$\mathcal{M}_{LST}$	$\mathcal{M}_{SK}$	$\mathcal{M}_G$	$\mathcal{M}_{CM}$
sound for int. check.	Yes	Yes	Yes	Yes	Yes
complete for int. check.	Yes	Yes	Yes	No	Yes
sound wrt. inc. tol.	Yes	Yes	Yes	No	Yes*
complete wrt. inc. tol.	Yes	weakly	weakly	No	No

* For singleton integrity theories

IV. Assessment of Integrity Checking Methods

As seen in section III, the differences between traditional integrity checking and ITIC are quite subtle. However, it would be wrong to think that inconsistency tolerance was for free or marginal. In this section, we assess five methods to determine if they *are* or *are not* inconsistency-tolerant, spanning from the seminal work by Nicolas [6] to more recent ones.

The result that many, though not all well-known methods are inconsistency-tolerant is of utmost practical significance, since each simplification method hitherto has been believed to be discapacitated, hence useless, in the presence of inconsistency. To show that several methods continue to function well even if integrity is violated thus breaks radically with all expectations. Without this result, there would be no justification at all for using integrity checking methods in inconsistent databases.

We chose methods [6], [5], [4] due to their impact on subsequent works. In particular, [6] initiated and popularized the notion of simplification. Its extensions in [5] and [4] have generalized integrity checking to datalog. A lot more extensions have appeared. Since it is unfeasible to discuss them all, we have chosen just two more methods. (Others are analyzed in [10].) One is from the 1990's [8]. It excels for constraints that lend themselves to optimizations related to query containment [11]. The other is from the 2000's [7]. It generates provably optimal simplifications, and generalizes previous methods that evaluate their simplifications in the old (instead of the new) state. Thus, costly rollbacks of updates that violate integrity are avoided. Table I summarizes the properties of the methods assessed in this section.

A. The Method of Nicolas

We are going to show that the well-known method for integrity checking by Nicolas [6], henceforth denoted by $\mathcal{M}_N$, is sound and complete wrt. inconsistency tolerance.

We adopt the notation $\Gamma^+_{f,I}$ from [6]. For a database D, a constraint $I = \vec{Q}I'$ in PCNF and a fact f to be inserted, $\mathcal{M}_N$ generates the simplification

$$\Gamma^+_{f,I} = \vec{Q}(I'\gamma_1 \wedge \cdots \wedge I'\gamma_m) \quad (m \geq 0) \tag{9}$$

where the γ_i are unifiers, restricted to *Glb*(I), of f and the m different occurrences of negated atoms in I unifying with f. Then, each occurrence of f in $\Gamma^+_{f,I}$ is replaced by *true* in $\Gamma^+_{f,I}$, which is then further simplified by standard rewritings. Symmetrically, for a fact f to be deleted, a simplification obtained by instantiating I with restricted unifiers of f and non-negated occurrences of matches of f, is generated. For simplicity, we only deal with insertions here; result and proof wrt. deletions are symmetrical.

Under the total integrity premise $D(I) = \mathit{true}$, the simplification theorem in [6] states that $D^U(I) = \mathit{true}$ iff $D^U(\Gamma^+_{f,I}) = \mathit{true}$.

***Example** 4.1:* Let I and U be as in Example 2.1. A PCNF of I is

$$\forall x \forall y \forall z\, (\sim p(x,y) \vee \sim p(x,z) \vee y = z)\,.$$

Clearly, $p(i,t)$ unifies with two atoms in I, by unifiers $\{x/i,\, y/t\}$ and $\{x/i,\, z/t\}$. The simplification $\Gamma^+_{p(i,t),I}$ returned by $\mathcal{M}_N$ is $\forall x \forall y \forall z\, (\sim p(i,t) \vee \sim p(i,z) \vee t{=}z) \wedge (\sim p(i,y) \vee \sim p(i,t) \vee y{=}t)$. Since the two conjuncts are obviously equivalent, one of them can be dropped, yielding $\forall x\ \forall y\ (\sim p(i,y) \vee \sim p(i,t) \vee y{=}t)$. Then, replacing $p(i,t)$ by *true* and dropping the corresponding disjunct yields the same simplification as in Example 2.1.

It is worth noting that each simplification step above is believed to be valid in [6] (and in fact in all the rest of the literature on integrity checking methods) only if I is satisfied in the old state. Theorem 4 rebuts this belief by confirming that the simplifications are valid also if I is violated in the old state. □

***Theorem** 4 (Inconsistency tolerance of $\mathcal{M}_N$):* $\mathcal{M}_N$ is sound and complete wrt. inconsistency tolerance in relational databases.

Proof: Let D be a relational database, ***IC*** an integrity theory and U an update. For the proof below, we assume that ***IC*** is singleton and $U = \mathit{insert}\ f$, for some fact $f \notin D$. A symmetric proof for deletions and straightforward extensions to multiple updates and constraints are omitted. In part *a)*, we show soundness, in *b)* completeness.

a) Soundness: Let $I = \vec{Q}I'$ be an integrity constraint in PCNF with matrix I', $\Gamma^+_{f,I}$ the simplification of $\mathcal{M}_N$ for I and U, σ a substitution such that *Dom*$(\sigma) =$ *Glb*(I), and $D(I\sigma) = \mathit{true}$. We have to show that $D^U(I\sigma) = \mathit{true}$ if $D^U(\Gamma^+_{f,I}) = \mathit{true}$. According to Theorem 2, it suffices to show:

$$\text{If } D^U(\Gamma^+_{f,I}) = \mathit{true} \text{ then } D^U(\Gamma^+_{f,I\sigma}) = \mathit{true}$$

where $\Gamma^+_{f,I\sigma}$ is the simplification of $I\sigma$ by $\mathcal{M}_N$. For each conjunct J in $\Gamma^+_{f,I\sigma}$, there is a negated atom g in I such that $g\sigma\beta = g\gamma = f$, where β and γ are the substitutions used to compute $\Gamma^+_{f,I\sigma}$ and, resp., $\Gamma^+_{f,I}$. Thus, $J = I'\sigma\beta = I'\gamma$ also occurs in $\Gamma^+_{f,I}$. Hence, $\Gamma^+_{f,I}$ entails $\Gamma^+_{f,I\sigma}$.

b) Completeness: We show the following contrapositive claim:

$$\begin{array}{l}\text{If } \mathcal{M}_N(D,I,U) = \mathit{vio} \text{ then there is a case } C \text{ of } I\\ \text{such that } D(C) = \mathit{true} \text{ and } D^U(C) = \mathit{false}\,.\end{array} \tag{10}$$

Assume $\mathcal{M}_N(D,I,U) = \mathit{vio}$. We distinguish $D(I) = \mathit{true}$ and $D(I) = \mathit{false}$. If $D(I) = \mathit{true}$, then $D^U(I) = \mathit{false}$, since $\mathcal{M}_N$ is complete for integrity checking. Hence, (10) follows.

Now, let $D(I) = \mathit{false}$. Since $\mathcal{M}_N$ reports a violation, there is a conjunct $I'\gamma$ in $\Gamma^+_{f,I}$ such that $C = \vec{Q}(I'\gamma)$ is violated in D^U, where γ is a unifier of f and a negative literal in I. Thus, one of the disjuncts in $I'\gamma$ is a negated occurrence of f. Since $f \notin D$, $D(C) = \mathit{true}$ holds. Hence (10) follows. ■

B. The Method of Lloyd, Sonenberg and Topor

We are going to show that the integrity checking method in [5], here denoted by $\mathcal{M}_{LST}$, is sound and weakly complete wrt. inconsistency tolerance.

In [5], two sets $\mathit{pos}_{D,D'}$ and $\mathit{neg}_{D,D'}$ are defined that capture the difference between any two databases D and D' such that $D \subseteq D'$. The sets consist of atoms that either are the head of some clause in the update $U = D' \setminus D$ or the head of a clause in D that is possibly affected by reasoning forward from clauses in U. In particular, $\mathit{pos}_{D,D'}$ captures a superset of the facts that are actually inserted, i.e., provable after the update but not before, and $\mathit{neg}_{D,D'}$ a superset of the facts that are actually deleted, i.e., provable before but not after the update.

Let D be a stratified database and U an update that preserves stratification. Applying the deletions in U to D leads to an intermediate state D''. Then, applying the insertions in U to D'' leads to the updated state $D' = D^U$. It is shown in [5] that $\mathit{pos}_{D'',D'} \cup \mathit{neg}_{D'',D}$ captures a superset of facts that are actually inserted, and $\mathit{neg}_{D'',D'} \cup \mathit{pos}_{D'',D}$ captures a superset of facts that are actually deleted by U.

Thus, the principles for identifying all relevant, i.e., potentially violated constraints, as established in [6], apply as follows. Only those atoms in $\mathit{pos}_{D'',D'} \cup \mathit{neg}_{D'',D}$ that unify with the atom of a negative literal in I by some mgu ϕ capture a possibly inserted fact that may violate integrity. That is checked by evaluating $\vec{Q}(I'\phi')$, where ϕ' is the restriction of ϕ to *Glb*(I), and I' the matrix of I. Symmetrically, only those atoms in $\mathit{neg}_{D'',D'} \cup \mathit{pos}_{D'',D}$ that unify with the atom of a positive literal in I by some mgu ϕ capture a possibly deleted fact that may violate integrity. That is then checked by evaluating the case $\vec{Q}(I'\phi')$ of I, where ϕ' is defined as above.

Let $\Phi(I,D,U)$ denote the set of all such substitutions ϕ' for identifying relevant constraints. Assuming the total integrity premise $D(I) = \mathit{true}$, the simplification theorem in [5] states that, for any stratified database D and update U preserving stratification, $D^U(I) = \mathit{true}$ iff, for all $\phi \in \Phi(I,D,U)$, $D^U(\vec{Q}(I'\phi)) = \mathit{true}$.

***Example** 4.2:* Let D consist of the four clauses

$$\begin{array}{ll} p(x,y) \leftarrow q(x) \wedge r(y), & r(b),\\ p(x,y) \leftarrow s(y,x), & s(b,a),\end{array}$$

$I = {\leftarrow} p(x,a)$ and $U = \{\mathit{insert}\ q(a)\}$. Clearly, $D(I) = \mathit{true}$. $\mathcal{M}_{LST}$ generates $\Phi(I,D,U) = \{x/a\}$, indicating that some fact matching $p(a,y)$ may violate I. Thus, the simplification to be evaluated is ${\leftarrow} p(a,a)$. Hence, $\mathcal{M}_{LST}(D,\{I\},U) = \mathit{sat}$. By the soundness of $\mathcal{M}_{LST}$, $D^U(I) = \mathit{true}$ follows.

Note that $\mathcal{M}_{LST}(D,\{I\},U) = \mathit{sat}$ also if $s(a,b) \in D$. That highlights the inconsistency tolerance of $\mathcal{M}_{LST}$, as stated below. □

***Theorem** 5 (Inconsistency tolerance of $\mathcal{M}_{LST}$):* $\mathcal{M}_{LST}$ is sound wrt. inconsistency tolerance in stratified databases.

Proof: Let D be a stratified database, $I = \vec{Q}I'$ be a constraint in PCNF with matrix I', $I^* = \vec{Q}(I'\zeta)$ a case of I, and U an update preserving stratification. Assume $D(I^*) = \mathit{true}$. We have to show that $D^U(I^*) = \mathit{true}$ if $D^U(\vec{Q}(I'\phi)) = \mathit{true}$, for all $\phi \in \Phi(I,D,U)$. That follows from (5) and lemma 4.1. ■

***Lemma** 4.1:* $D^U(\vec{Q}(I'\zeta\phi^*)) = \mathit{true}$ for all $\phi^* \in \Phi(I^*,D,U)$ if $D^U(\vec{Q}(I'\phi)) = \mathit{true}$ for all $\phi \in \Phi(I,D,U)$.

This lemma is a direct consequence of the following one.

***Lemma** 4.2:* For each substitution $\phi^* \in \Phi(I^*,D,U)$ there is a substitution $\phi \in \Phi(I,D,U)$ that is more general than ϕ^*.

Proof: Each $\phi^* \in \Phi(I^*, D, U)$ either originates from a potentially inserted atom A that unifies with a negated atom A^* in I^* or a potentially deleted atom B that unifies with a non-negated atom B^* in I^*. Since I^* is a case of I, there is a negated atom A' (or a non-negated atom B', resp.) in I such that $A^* = A'\zeta$ ($B^* = B'\zeta$, resp.). Thus, A (B, resp.) *a fortiori* unifies with A' (B', resp.), by some mgu ϕ that is more general than ϕ^*. ■

The method $\mathcal{M}_{LST}$ is not complete wrt. inconsistency tolerance, as shown by the following counter-example.

Example 4.3: For the same D, I and U as in Example 4.2, let $D^* = D \cup \{s(a,a)\}$. Clearly, $\leftarrow p(a,a)$ is a violated case in D^*; all other basic cases of I are satisfied in D^*. Although U does not introduce any new violated case in $(D^*)^U$, $\mathcal{M}_{LST}$ still generates the simplification $\leftarrow p(a,a)$. Thus, $\mathcal{M}_{LST}(D^*, \{I\}, U) = \mathit{vio}$. Hence, by Def. 3.2, $\mathcal{M}_{LST}$ is not complete wrt. inconsistency tolerance. Yet, we have the following result. □

Theorem 6 *(Weak completeness of $\mathcal{M}_{LST}$):* $\mathcal{M}_{LST}$ is weakly complete wrt. inconsistency tolerance in stratified databases.

Proof: Let D be a stratified database, *IC* an integrity theory and U an update. We have to show that $D^U(\mathit{IC}) = \mathit{true}$ entails $\mathcal{M}_{LST}(D, \mathit{IC}, U) = \mathit{sat}$. If $D^U(\mathit{IC}) = \mathit{true}$, then there is a refutation of $\leftarrow I'$ in D^U for each case I' of each constraint I in *IC*. Since each simplification of I checked by $\mathcal{M}_{LST}$ is a case of I, it follows that $\mathcal{M}_{LST}(D, \mathit{IC}, U) = \mathit{sat}$. ■

C. The Method of Sadri & Kowalski

We are going to show that the integrity checking method in [4], here denoted by $\mathcal{M}_{SK}$, is sound and weakly complete wrt. inconsistency tolerance.

Roughly, $\mathcal{M}_{SK}$ works as follows. Each integrity theory IC is a set of denials. Each update U may include denials. Denials to be deleted cannot violate integrity and thus are simply dropped from *IC*: let $\mathit{IC}^- = \mathit{IC} \setminus \{I : \mathit{delete}\, I \in U\}$. Denials to be inserted are queried in the new state. If any of them is refuted, $\mathcal{M}_{SK}$ outputs *vio*. Else, for checking if any other clause in U would cause integrity violation, $\mathcal{M}_{SK}$ computes an update U', consisting of all clauses in U to be inserted and all ground negative literals $\sim H$ such that H is *true* in D and *false* in D^U. For each $T \in U'$, $\mathcal{M}_{SK}$ builds a resolution tree rooted at T, using input clauses from $D^U \cup \mathit{IC}^-$. For each derivation δ in the tree, each step taken in δ is either a standard backward-reasoning step, or a forward-reasoning step from the literal selected in the head of T or of any clause derived from T by previous steps in δ. In forward steps, the selected literal is resolved with a matching literal in the body of some input clause. If any such derivation yields a refutation, $\mathcal{M}_{SK}$ outputs *vio*. If the tree is finitely failed, $\mathcal{M}_{SK}$ outputs *sat*.

Theorem 7 *(Inconsistency tolerance of $\mathcal{M}_{SK}$):* $\mathcal{M}_{SK}$ is sound wrt. inconsistency tolerance in stratified databases.

Proof: Let D be a stratified database, *IC* an integrity theory and U an update such that $\mathcal{M}_{SK}(D, \mathit{IC}, U) = \mathit{sat}$. Further, let IC^- and U' be as described above. We have to show that, for each $C \in \mathsf{S}(D, \mathit{IC}^-)$, $D^U(C) = \mathit{true}$. By Theorem 2, it suffices to verify (5), i.e. that for each $C_0 \in U'$, $\mathcal{M}_{SK}$ builds a finitely failed tree $\mathcal{T}_C$, rooted at C_0, with input from $D^U \cup \{C\}$.

Let $C \in \mathsf{S}(D, \mathit{IC}^-)$, i.e., for some $I \in \mathit{IC}^-$, $C = I\sigma$, for some substitution σ of *Glb*(I). Further, let $C_0 \in U'$. Since $\mathcal{M}_{SK}(D, \mathit{IC}^-, U) = \mathit{sat}$, there is a finitely failed tree $\mathcal{T}$ in the search space of $\mathcal{M}_{SK}$, rooted at C_0, built with input clauses from $D^U \cup \mathit{IC}^-$. From $\mathcal{T}$, $\mathcal{T}_C$ is obtained as follows.

Each derivation δ in $\mathcal{T}$ is replaced, if possible, by the following derivation δ' in $\mathcal{T}_C$. It starts from the same root as δ. For each i, $0 \le i < n$, where n is the length of δ, the i+1-th resolvent of δ' is obtained as follows. Suppose the j-th literal of the i-th clause of δ is selected. Then, also the j-th literal in the i-th clause of δ' is selected. If the i+1-th input clause of δ is I, then C is used as input clause in δ'; if the k-th literal is selected in I, then also the k-th literal is selected in C. Otherwise, the i+1-th input clause of δ is also used in δ', for obtaining the i+1-th resolvent of δ'. Clearly, the latter is of form $C_{i+1}\sigma_{i+1}$, for some substitution σ_{i+1}, where C_{i+1} is the i+1-th resolvent in δ.

At any step of δ', it may be impossible to continue its construction by using the input clause corresponding to the one used in δ: either the selected literal does not match with the selected literal in the corresponding input clause in δ, or the latter is a denial which cannot be used as input for $\mathcal{T}_C$. In both cases, δ' is discontinued, i.e., δ' then terminates with failure.

It is easy to see that $\mathcal{T}_C$ is the required finitely failed tree. ■

We illustrate the inconsistency tolerance of $\mathcal{M}_{SK}$ with an example inspired by a similar one in [12].

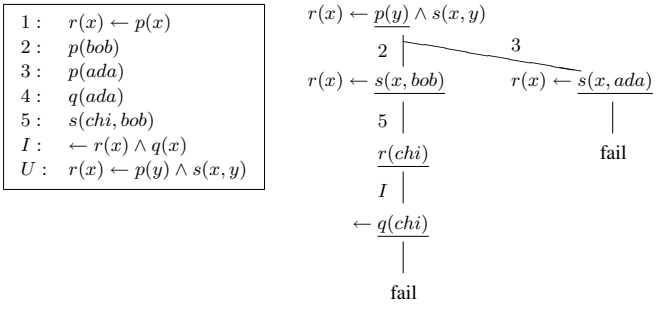

Fig. 1. Clauses and derivation tree of Example 4.4.

Example 4.4: Let D be a database consisting of clauses 1–5 in Figure 1, defining the predicates r (regular), p (pays taxes), q (quitted) and s (signed). The integrity constraint I denies the possibility to have regular status and to have quitted work at the same time. The update U inserts a clause stating that persons signed by a tax payer also have regular status.

Clearly, $D(I) = \mathit{false}$, since $r(ada)$ and $q(ada)$ are *true* in D. The case $I' = \leftarrow r(bob) \wedge q(bob)$ of I, however, is satisfied in D since $q(bob)$ is not *true* in D.

From the root U, $\mathcal{M}_{SK}$ builds the tree as shown in Figure 1 (selected literals are underlined). Since this tree is finitely failed, it follows that U will not introduce new cases of inconsistency: all cases of integrity constraints that were satisfied in D remain satisfied in D^U. In particular, I' is also satisfied in D^U. □

The method $\mathcal{M}_{SK}$ is not complete wrt. inconsistency tolerance, as shown by the following counter-example.

Example 4.5: Let D^*, U, and *IC* be as in Example 4.3, and $D^{**} = D^* \cup \{r(a)\}$. The only violated basic case in D^{**} is $\leftarrow p(a,a)$, and U does not introduce any additional one. However, starting from U, $\mathcal{M}_{SK}$ derives $p(a,y) \leftarrow r(y)$, which it then refutes by two more steps for resolving the literals in head and body against $\leftarrow p(a,a)$ and $r(a)$. Thus, $\mathcal{M}_{SK}$ is not complete wrt. inconsistency tolerance. Yet, we have the following result. □

Theorem 8 *(Weak completeness of $\mathcal{M}_{SK}$):* $\mathcal{M}_{SK}$ is weakly complete wrt. inconsistency tolerance in stratified databases.

Proof: Let D be a stratified database, *IC* an integrity theory and U an update for which $\mathcal{M}_{SK}$ terminates. We have to show that $D^U(\mathit{IC}) = \mathit{true}$ entails $\mathcal{M}_{SK}(D, \mathit{IC}, U) = \mathit{sat}$. Suppose that

$\mathcal{M}_{SK}(D, \mathit{IC}, U) = \mathit{vio}$. Thus, by definition of $\mathcal{M}_{SK}$, there is a refutation rooted at some clause in U with input clauses from D^U plus a denial clause in *IC*. Hence, integrity is violated in D^U. However, this contradicts the supposition above. Since $\mathcal{M}_{SK}$ terminates, the result follows. ■

D. The Method of Gupta, Sagiv, Ullman and Widom

Not all methods are inconsistency-tolerant. An example is the well-known method in [8], here denoted by $\mathcal{M}_G$. The constraints considered in [8] are of the form

$$\leftarrow L \wedge R_1 \wedge \cdots \wedge R_n \wedge E_1 \wedge \cdots \wedge E_k \tag{11}$$

where L is a literal with a *local*, i.e., accessible predicate; the R_i are literals with *remote* predicates that are not accessible for integrity checking; the E_j are evaluable literals with arithmetic expressions. Updates considered in [8] are insertions of facts into the relation of L. (In fact, the method also works if L is a conjunction of literals.) For convenience, let l be L's predicate.

The main result, Theorem 5.2 in [8], refers to a simplification called *reduction*. For a constraint I of the form (11) and a fact f inserted in l, the reduction $\mathit{RED}(f, L, I)$ is essentially the corresponding simplification in $\mathcal{M}_N$. To check if I is satisfied after inserting f, $\mathcal{M}_G$ checks if, for facts g in the extension of l,

$$\mathit{RED}(f, L, I) \sqsubseteq \bigcup_{g \ in \ l} \mathit{RED}(g, L, I)$$

holds, where $\sqsubseteq$ denotes query containment.

***Example** 4.6:* The constraint $I = \leftarrow l(x, y) \wedge r(z) \wedge x \leq z \leq y$ requires that no z in r must occur in an interval whose ends are specified by l. Suppose $D = \{l(3, 6), l(5, 10)\}$ and U inserts $l(4, 8)$. Then, $D^U(I) = \mathit{true}$ is inferred by $\mathcal{M}_G$ from

$$r(z) \wedge 4 \leq z \leq 8 \ \sqsubseteq \ (r(z) \wedge 3 \leq z \leq 6) \cup (r(z) \wedge 5 \leq z \leq 10),$$

which essentially expresses that $[4, 8]$ is contained in $[3, 10]$. □

Example 4.7 shows that $\mathcal{M}_G$ is not inconsistency-tolerant.

***Example** 4.7:* Consider $D = \{l(3, 6), l(5, 10), r(7)\}$, the case $I' = \leftarrow l(4, 8) \wedge r(z) \wedge 4 \leq z \leq 8$ of I and also U as in Example 4.6. Clearly, $D(I) = \mathit{false}$, while $D(I') = \mathit{true}$. Assuming the total integrity premise, $\mathcal{M}_G$ guarantees, as in 4.6, that U does not violate integrity, i.e., $\mathcal{M}_G(D, \{I\}, U) = \mathit{sat}$. However, $D^U(I') = \mathit{false}$. Thus $\mathcal{M}_G$ is not sound wrt. inconsistency tolerance. □

As reported in [8], $\mathcal{M}_G$ cannot be complete for integrity checking since constraints may involve inaccessible remote data. Thus, by Theorem 1, $\mathcal{M}_G$ is not complete wrt. inconsistency tolerance either, nor is it weakly complete, by Corollary 3*b)*.

E. The Method of Christiansen and Martinenghi

For an integrity theory *IC* and an update U, the method in [7], here dentoted by $\mathcal{M}_{CM}$, consists of the following two steps:

- First, a "pre-simplification" of *IC* for U, denoted $\mathsf{After}^U(\mathit{IC})$, is computed, as described in Def. 4.3 below, such that $D(\mathsf{After}^U(\mathit{IC})) = D^U(\mathit{IC})$, for every database D.
- Second, $\mathsf{After}^U(\mathit{IC})$ is optimized by removing from it all denials and literals that can be proved to be redundant, assuming that the total integrity premise, i.e., $D(\mathit{IC}) = \mathit{true}$, holds. The result is denoted $\mathsf{Optimize}^{IC}(\mathsf{After}^U(\mathit{IC}))$.

To run $\mathcal{M}_{CM}(D, \mathit{IC}, U)$ is to compute the simplification $\mathsf{Optimize}^{IC}(\mathsf{After}^U(\mathit{IC}))$ and evaluate it in D.

***Definition** 4.3: For an integrity theory IC and an update U, let* $\mathsf{After}^U(\mathit{IC})$ *be obtained from IC by simultaneously replacing each atom of the form* $p(\vec{t})$ *by* $(p(\vec{t}) \wedge \vec{t} \neq \vec{b}_1 \wedge \cdots \wedge \vec{t} \neq \vec{b}_m) \vee \vec{t} = \vec{a}_1 \vee \cdots \vee \vec{t} = \vec{a}_n$*, where* $p(\vec{a}_1), \ldots, p(\vec{a}_n)$ *are all facts inserted to p and* $p(\vec{b}_1), \ldots, p(\vec{b}_m)$ *are all facts deleted from p by U.*

By using De Morgan's laws, $\mathsf{After}^U(\mathit{IC})$ is represented as a set of denials. Then, $\mathsf{Optimize}^{IC}(\mathsf{After}^U(\mathit{IC}))$ is obtained, as specified in Definition 4.4, by a terminating proof procedure, denoted below by $\vdash$, that is *substitutive*[1], i.e., if $\mathcal{F} \vdash \mathcal{F}'$ then $\mathcal{F}\sigma \vdash \mathcal{F}'\sigma$, for each pair $\mathcal{F}$, $\mathcal{F}'$ of sets of formulas and each substitution σ.

***Definition** 4.4: Let IC, IC′ be sets of denials, I a denial, K a conjunction of literals, L a literal, and* $\vdash$ *a terminating proof procedure.* $\mathsf{Optimize}^{IC}(\mathit{IC}')$ *is obtained by exhaustively applying on IC′ the following rewrite rules, where* $\mathit{IC}'' = \mathit{IC}' \setminus \{\leftarrow K \wedge L\}$.

1) $\mathit{IC}' \rightsquigarrow \mathit{IC}'' \cup \{\leftarrow K\}$ *if* $\leftarrow K \wedge L \in \mathit{IC}'$ *and* $\mathit{IC} \cup \mathit{IC}' \vdash \leftarrow K$

2) $\mathit{IC}' \rightsquigarrow \mathit{IC}' \setminus \{I\}$ *if* $I \in \mathit{IC}'$ *and* $\mathit{IC} \cup (\mathit{IC}' \setminus \{I\}) \vdash I$

In [7] it is shown that $\mathcal{M}_{CM}$ is both sound and complete.

***Example** 4.8:* Let I and U be as in Example 3.2. We have

$$\begin{aligned}\mathsf{After}^U(\{I\}) = \{ \ & \leftarrow p(x, y) \wedge p(x, z) \wedge y \neq z, \\ & \leftarrow x = i \wedge y = t \wedge p(x, z) \wedge y \neq z, \\ & \leftarrow p(x, y) \wedge x = i \wedge z = t \wedge y \neq z, \\ & \leftarrow x = i \wedge y = t \wedge x = i \wedge z = t \wedge y \neq z\}\end{aligned}$$

Then, Optimize removes the first constraint (subsumed by I), the second (subsumed by the third), and the fourth (a tautology). The simplification returned by $\mathcal{M}_{CM}$ (to be evaluated in the old state) is the third constraint, equivalent to I' as found in Example 3.2. Then, for each database D, $D^U(I) = \mathit{true}$ iff $D(I') = \mathit{true}$. □

The $\mathcal{M}_{CM}$ method is not sound wrt. inconsistency tolerance due to the behavior of Optimize, as illustrated in Example 4.9.

***Example** 4.9:* Let $\mathit{IC} = \{\leftarrow p \wedge q, \leftarrow p \wedge \sim q, \leftarrow p \wedge r(x) \wedge s(x)\}$ and $U = \{\mathit{insert}\ r(a)\}$. The simplification IC' of *IC* for U computed by $\mathcal{M}_{CM}$ is $\emptyset$ (i.e., U cannot violate integrity if $D(\mathit{IC}) = \mathit{true}$), since $\leftarrow p$, derived from *IC* by $\vdash$, subsumes all denials in $\mathsf{After}^U(\mathit{IC}) = \{\leftarrow p \wedge q, \leftarrow p \wedge \sim q, \leftarrow p \wedge r(x) \wedge s(x), \leftarrow p \wedge s(a)\}$.

Now, let $D = \{p, s(a)\}$. Clearly, $I = \leftarrow p \wedge r(a) \wedge s(a)$ is a case of the last constraint in *IC*. We have: $D(\mathit{IC}) = \mathit{false}$, $D(I) = \mathit{true}$ and $D(\mathit{IC}') = \mathit{true}$. However, $D^U(I) = \mathit{false}$, which shows that $\mathcal{M}_{CM}$ is not sound wrt. inconsistency tolerance. □

One may object that *IC* above is equivalent to $\leftarrow p$ and thus redundant. For $\mathit{IC} = \{\leftarrow p\}$, misleading optimizations would be avoided. In general, however, redundancy is undecidable.

Optimize never harms inconsistency tolerance if *IC* contains a single constraint, as shown by Theorem 9 below. (More generally, it can be shown that $\mathcal{M}_{CM}$ is sound wrt. inconsistency tolerance if each pair of constraints has no predicate in common.)

***Theorem** 9 (Inconsistency tolerance of $\mathcal{M}_{CM}$):* For singleton integrity theories, $\mathcal{M}_{CM}$ is sound wrt. inconsistency tolerance in hierarchical databases.

Proof: Let D be a hierarchical database, I a denial, U an update, I' the simplification of $\{I\}$ for U obtained by $\mathcal{M}_{CM}$ and θ a substitution such that $I\theta \in \mathsf{S}(D, I)$. Since $\mathcal{M}(D, \{I\}, U) = D(I')$, we have to show:

$$\text{If } D(I') = \mathit{true} \text{ then } D^U(I\theta) = \mathit{true}. \tag{12}$$

We prove (12) by transitivity of (13) and (14), below, as follows.

$$\text{If } D(I') = \mathit{true} \text{ then } D(I'\theta) = \mathit{true}. \tag{13}$$

$$\text{If } D(I'\theta) = \mathit{true} \text{ then } D^U(I\theta) = \mathit{true}. \tag{14}$$

[1] Substitutivity of $\vdash$ is assumed implicitly in [7].

Evidently, (13) holds. Note now that, by Definition 4.3, After is substitutive, i.e., $\mathsf{After}^U(I\theta) = \mathsf{After}^U(I)\theta$. Since $\vdash$ is also substitutive, $I'\theta$ is obtained from $\mathsf{After}^U(I\theta)$ by the same sequence of Optimize steps as I' is obtained from $\mathsf{After}^U(I)$. Then, (14) holds because, as shown in [7] to prove soundness of $\mathcal{M}_{CM}$, the evaluation of the result of After in the old state is a sound integrity checking method, and the application of any of the steps in Optimize preserves soundness. ■

The interplay between multiple constraints also causes $\mathcal{M}_{CM}$ to be not complete (not even weakly) wrt. inconsistency tolerance. This is shown by the following counter-example.

***Example** 4.10:* For $D = \{q(b)\}$, $IC = \{\leftarrow p(a) \wedge q(x),\ \leftarrow q(b)\}$ and U = {*insert* $p(a)$, *delete* $q(b)$}, we obtain $\mathsf{After}^U(IC) = \{\leftarrow q(x) \wedge a{=}a \wedge x{\neq}b,\ \leftarrow p(a) \wedge q(x) \wedge x{\neq}b,\ \leftarrow q(b) \wedge b{\neq}b\}$. From that, $\mathsf{Optimize}^{IC}(\mathsf{After}^U(IC)) = \{\leftarrow q(x)\}$ is obtained as follows. First, the third denial in $\mathsf{After}^U(IC)$ is dropped, since it is subsumed by the second denial in *IC*. Then, $a = a$ is dropped in the first denial. That then subsumes the second denial, which is thus removed. Last, $x \neq b$ is dropped from the remaining denial $\leftarrow q(x) \wedge x \neq b$, since $\leftarrow q(x)$ can be proved from $\leftarrow q(x) \wedge x \neq b$ and $\leftarrow q(b)$ in *IC*. Thus, since $D^U(IC) = \mathit{true}$ and $\mathcal{M}_{CM}(D, IC, U) = D(\mathsf{Optimize}^{IC}(\mathsf{After}^U(IC))) = \mathit{false}$, neither (4) nor (6) holds for $\mathcal{M}_{CM}$. □

V. Applications

Inconsistency-tolerant integrity checking can improve solutions to several problems of database management. We show that for update requests in V-A, for repairs in V-B, for schema evolution in V-C, for reliable risk management in V-D, and for unsatisfiability handling in V-E.

Updates are a cornerstone of each database management application addressed in this section. Each update U is required to preserve the satisfaction of a given integrity theory *IC*. Traditionally, integrity preservation has meant that U maps a state D such that $D(IC) = \mathit{true}$ to a state D^U such that $D^U(IC) = \mathit{true}$. But as soon as constraint violations in D become tolerable, the notion of integrity preservation must be generalized as follows.

***Definition** 5.1: For a database D and an integrity theory IC, an update U is said to* preserve integrity *if* $\mathsf{S}(D, IC) \subseteq \mathsf{S}(D^U, IC)$.

Note that definition 5.1 does not require total integrity of D, i.e., U may preserve integrity even if executed in the presence of violated constraints. The following corollary of Definitions 3.2 and 5.1 states that updates can be checked for preserving integrity by any method that is sound wrt. inconsistency tolerance.

***Corollary** 10:* For a database D, an integrity theory *IC* and an inconsistency-tolerant integrity checking method $\mathcal{M}$, an update U preserves integrity if $\mathcal{M}(D, IC, U) = \mathit{sat}$.

In general, the only-if half of Corollary 10 does not hold, as shown in Example 5.1. It is easily seen that it does hold for methods that are complete wrt. inconsistency tolerance.

***Example** 5.1:* Let p be defined by $p(x, y) \leftarrow s(x, y, z)$ and $p(x, y) \leftarrow q(x) \wedge r(x, y)$ in a database D in which $q(a)$ and $r(a, a)$ are the only facts that contribute to the natural join of q and r. Further, let $IC = \{\leftarrow p(x, x)\}$ and $U = \{\mathit{insert}\ s(a, a, b)\}$. Clearly, U preserves integrity, since the case $C = \leftarrow p(a, a)$ is already violated in D. However, the inconsistency-tolerant methods $\mathcal{M}_{LST}$ and $\mathcal{M}_{SK}$ and others generate and evaluate the simplification $\leftarrow p(a, a)$ of $\leftarrow p(x, x)$ and thus output *vio*. □

A. Inconsistency-tolerant Satisfaction of Update Requests

We define an *update request* as a closed first-order formula intended to be made *true* by some integrity-preserving update. For a database D, an update U is said to *satisfy* an update request R if $D^U(R) = \mathit{true}$ and U preserves integrity. 'View update' requests are a common variant of update requests. An *update method* is a method to compute updates for satisfying update requests.

Similar to integrity checking, also all known update methods have traditionally postulated the total satisfaction of all constraints in the old state. However, that requirement is as unrealistic for satisfying update requests as for integrity checking. And, in fact, we are going to see that it can be abandoned just as well, for the class of methods defined as follows.

***Definition** 5.2: An update method $\mathcal{UM}$ is* inconsistency-tolerant *if each update computed by $\mathcal{UM}$ preserves integrity.*

For an update request R and a database D, many update methods work in two phases. First, an update U such that $D^U(R) = \mathit{true}$ is computed. Then, U is checked for integrity preservation by some integrity checking method. If that check is positive, U is accepted. Else, U is rejected and another update candidate, if any, is computed and checked. Hence, the following corollary follows from Definition 5.2 and Corollary 10.

***Corollary** 11:* Each update method that uses an inconsistency-tolerant method to check its computed updates for preserving integrity is inconsistency-tolerant.

Corollary 11 serves to identify several known update methods as inconsistency-tolerant, since they use inconsistency-tolerant integrity checking methods. Among them are, e.g., the update methods in [13], [14] which use the integrity checking method of [5], shown to be inconsistency-tolerant in IV-B.

Another well-known update method, by Kakas & Mancarella, is described in [15]. For convenience, let us name it $\mathcal{KM}$. It does not use any integrity checking method as a separate module, hence Corollary 11 is not applicable. However, the inconsistency tolerance of $\mathcal{KM}$ can be tracked down as outlined below.

For satisfying an update request, $\mathcal{KM}$ explores a possibly nested search space of 'abductive' derivations and 'consistency' derivations. Roughly, abductive derivations compute hypothetical updates of facts for satisfying a given update request; consistency derivations check these updates for integrity. Each update generated by $\mathcal{KM}$ consists of a bipartite set of positive and negative ground literals, corresponding to insertions and, resp., deletions of ground facts. For more details, we refer the reader to [15]. It suffices here to mention that, for $\mathcal{KM}$, all constraints are represented by denials that are used as candidate input clauses in consistency derivations. Each consistency derivation of each update computed by $\mathcal{KM}$ corresponds to a finitely failed attempt to refute the update as inconsistent.

It is easy to verify that, for an update request R, each update U computed by $\mathcal{KM}$ makes R become *true* in D^U, even if some constraint is violated in D. What is at stake is the preservation of the satisfaction of each case that is satisfied in D, while cases that are violated in D may remain violated in D^U. The following theorem entails that satisfied cases are preserved by $\mathcal{KM}$.

***Theorem** 12:* The method $\mathcal{KM}$ is inconsistency-tolerant.

Proof: By Definition 5.2, we have to show that each update computed by $\mathcal{KM}$ preserves integrity. Suppose that, for some update request in some database D and some integrity theory *IC*, $\mathcal{KM}$ would compute and accept an update U that does not preserve integrity. Then, by Definition 5.1, there is a constraint I

in *IC* such that $D(I) = \mathit{true}$ and $D^U(I) = \mathit{false}$. Hence, by the definition of $\mathcal{KM}$, there is a consistency derivation δ rooted at one of the literals in U, that uses I as input clause and terminates by deducing the empty clause. That, however, signals that the root of δ causes a violation of I. Thus, $\mathcal{KM}$ rejects U, which contradicts the supposition that $\mathcal{KM}$ accepts U. ■

Example 5.2 illustrates the usefulness of inconsistency-tolerant update methods.

***Example** 5.2:* Let $D = \{q(x) \leftarrow r(x) \wedge s(x),\ \ p(a,a)\}$, *IC* = $\{\leftarrow p(x,x),\ \leftarrow p(a,y) \wedge q(y)\}$ and R the update request to make $q(a)$ *true*. To satisfy R, most update methods compute the candidate update $U = \{\mathit{insert}\ r(a),\ \mathit{insert}\ s(a)\}$. To check if U preserves integrity, most integrity checking methods compute the simplification $\leftarrow p(a,a)$ of the second constraint in *IC*. Rather than accessing the p relation for evaluating $\leftarrow p(a,a)$, integrity checking methods that are not inconsistency-tolerant may use the invalid total integrity premise that $D(\mathit{IC}) = \mathit{true}$, by reasoning as follows. The first constraint $\leftarrow p(x,x)$ in *IC* is not affected by U and subsumes $\leftarrow p(a,a)$, hence both constraints remain satisfied in D^U. Thus, such methods conclude that U preserves integrity. However, that is wrong, since the case $\leftarrow p(a,y) \wedge q(y)$ is satisfied in D but violated in D^U. By contrast, each inconsistency-tolerant update method rejects U and computes the update $U' = U \cup \{\mathit{delete}\ p(a,a)\}$ for satisfying R. Clearly, U' preserves integrity. Incidentally, U' even removes a violated case. □

B. Partial Repairs

Roughly, 'repairing' is to compute updates to databases with violated constraints such that the updated databases satisfy integrity. Based on cases, Definition 5.3 introduces 'partial repairs'. They repair only a fragment of the database.

***Definition** 5.3: Let D be a database, IC an integrity theory and S a set of cases of constraints in 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 $D^U(\mathit{IC}) = \mathit{false}$, U is also called a* partial repair *of IC in D, else U is called a* total repair *of IC in D.*

Related notions in the literature [1], [16], [17] only deal with total repairs, additionally requiring them to be minimal, in some sense. In [18], null values and a 3-valued semantics are used to "summarize" total repairs.

Repairing can be costly, if not intractable [19]. Thus, at first sight, a good heuristic to curtail inconsistency could be to use partial instead of total repairs, particularly in large databases with potentially unknown inconsistencies. However, partial repairs may not preserve integrity, as shown by the following example.

***Example** 5.3:* Let $\mathit{IC} = \{\leftarrow p(x,y,z) \wedge {\sim}q(x,z),\ \ \leftarrow q(x,x)\}$ and $D = \{p(a,b,c), p(b,b,c), p(c,b,c), q(a,c), q(c,c)\}$. The violated basic cases are $\leftarrow p(b,b,c) \wedge {\sim}q(b,c)$ and $\leftarrow q(c,c)$. Repairing $\{\leftarrow q(x,x)\}$ by $\{\mathit{delete}\ q(c,c)\}$ does not preserve integrity, since $\leftarrow p(c,b,c) \wedge {\sim}q(c,c)$ is satisfied in D but not in D^U. However, the partial repairs $\{\mathit{delete}\ p(b,b,c)\}$ and $\{\mathit{insert}\ q(b,c)\}$ of *IC* do preserve integrity. The only subset-minimal total repairs are $\{\mathit{delete}\ q(c,c),\ \mathit{delete}\ p(b,b,c),\ \mathit{delete}\ p(c,b,c)\}$ and $\{\mathit{delete}\ q(c,c),\ \mathit{insert}\ q(b,c),\ \mathit{delete}\ p(c,b,c)\}$. □

The dilemma that total repairs may require more update operations than partial repairs, while the latter may not preserve integrity, is relaxed by the following corollary of Corollary 10. It says that it suffices to check partial repairs for integrity preservation by an inconsistency-tolerant method.

***Corollary** 13:* For each inconsistency-tolerant method $\mathcal{M}$, each database D and each integrity theory *IC*, each partial repair U of *IC* such that $\mathcal{M}(D, \mathit{IC}, U) = \mathit{sat}$ preserves integrity.

The following example illustrates how integrity-preserving repairs can be computed by inconsistency-tolerant update methods.

***Example** 5.4:* Let D be a database and $S = \{\leftarrow B_1, \ldots, \leftarrow B_n\}$ ($n \geq 0$) a set of cases of constraints in an integrity theory *IC*. Thus, $D(S) = \mathit{false}$ if and only if $D(\leftarrow B_i) = \mathit{true}$ for some i. Hence, an integrity-preserving repair of S that tolerates extant violations of cases not in S can be computed by each inconsistency-tolerant update method, by issuing the update request ${\sim}\mathit{vio}_S$, where vio_S is defined by the n clauses $\mathit{vio}_S \leftarrow B_1, \ldots, \mathit{vio}_S \leftarrow B_n$. Update methods that are not inconsistency-tolerant cannot be used, since they may accept repairs that do not preserve integrity, as seen in Example 5.2. □

C. Inconsistency-tolerant Schema Evolution

A database schema evolves via *schema updates*, i.e., removals, additions or alterations of integrity constraints or of database clauses with non-empty bodies. Since changes of the set of clauses can be captured by update requests as in V-A, and deletions of constraints never cause any violation, we focus below on schema updates consisting of insertions of constraints.

Whenever a new constraint I is added to the integrity theory, it may be too costly to evaluate it on the spot, let alone to immediately repair all violated cases of I. As long as such repairs are delayed, traditional integrity checking is not applicable, since the total integrity premise does not hold. However, inconsistency-tolerant integrity checking can be used, no matter for how long the repair of violated cases is delayed.

More precisely, let $\mathit{IC}^* = \mathit{IC} \cup \{I\}$ be an integrity theory obtained by the schema update *insert* I. Then, each inconsistency-tolerant method $\mathcal{M}$ for computing $\mathcal{M}(D, \mathit{IC}^*, U)$ for each update U issued after *IC* has been updated can guarantee that all cases in IC^* that are satisfied in D remain satisfied in D^U. If $\mathcal{M}$ was not inconsistency-tolerant, then a possible inconsistency of $D \cup \mathit{IC}^*$ would invalidate any output of $\mathcal{M}(D, \mathit{IC}^*, U)$, even if integrity was totally satisfied before *IC* was updated.

Theorem 14 below captures another advantage of inconsistency-tolerant integrity checking for schema evolution.

***Theorem** 14:* For each database D, each pair of integrity theories *IC*, *IC*′, each update U and each inconsistency-tolerant method $\mathcal{M}$, the following holds, where $\mathit{IC}^* = \mathit{IC} \cup \mathit{IC}'$.

$$\text{If } D(\mathit{IC}){=}\mathit{true} \text{ and } \mathcal{M}(D, \mathit{IC}^*, U){=}\mathit{sat} \text{ then } D^U(\mathit{IC}){=}\mathit{true} \quad (15)$$

Proof: Since $\mathit{IC} \subseteq \mathsf{S}(D, \mathit{IC}^*)$, (15) follows from (3). ■

Theorem 14 says that $\mathcal{M}$ guarantees the preservation of total integrity of *IC* even if $D(\mathit{IC}') = \mathit{false}$. That is useful for distinguishing *hard* constraints (those in *IC*), the satisfaction of which is indispensable, and *soft* constraints (in *IC*′), the violation of which is tolerable. Thus, by Theorem 14, each inconsistency-tolerant method guarantees that all hard constraints remain totally satisfied across updates even if there are violated soft constraints.

***Example** 5.5:* Let *hr* and *lr* be two predicates that model a high, resp., low risk in some application domain. Further, $I_1 = \leftarrow hr(\vec{x})$, $I_2 = \leftarrow lr(\vec{x})$, be a hard, resp., soft constraint for protecting against high and, resp., low risks. Then, each inconsistency-tolerant method $\mathcal{M}$ can be used to preserve the satisfaction of I_1 across updates, even if I_2 is violated. □

D. Inconsistency-tolerant Risk Management

Since constraint violations may be hidden or unknown, and since all integrity checking methods traditionally have insisted on total integrity, their use has not been reliable. But now, the definition of inconsistency-tolerant integrity checking provides a decision criterion for distinguishing reliable from unreliable methods. The unreliability of methods that are not inconsistency-tolerant is illustrated in the following elaboration of example 5.5.

***Example** 5.6:* Let $D = \{p(0,0), p(1,2), p(2,3), p(3,4), ...\}$. Further, let the predicates in $IC = \{\leftarrow lr(x), \leftarrow hr(x)\}$ be defined by the clauses

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

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

where *lr* and *hr* indicate a low and, resp., a high risk. In the clause defining *hr*, the term *th* may stand for a threshold value.

The purpose of *IC* is to protect the application from any risk. Yet, in D, the low-risk presence of $p(0,0)$ is tolerated. Now, let U = *insert* $q(0, 100000)$. Then, methods that are not inconsistency-tolerant, such as $\mathcal{M}_G$, $\mathcal{M}_{CM}$, reason as follows for checking if U preserves integrity. Using U for simplifying $\leftarrow hr(x)$ yields the case $\leftarrow p(0,0), 100000 > th$. It is obtained from the body of the definition of *hr* by binding the variables x and y to 0 and, resp., 100000, and then dropping the literal $q(0, 100000)$. Clearly, that case is subsumed by $\leftarrow p(x,x)$, which defines *lr* and is not affected by U. The total integrity premise entails that $\leftarrow p(x,x)$ is satisfied in D. Hence, methods that are not inconsistency-tolerant may deduce that $\leftarrow p(x,x)$ remains satisfied in D^U. From that, such methods deduce that also the subsumed constraint $\leftarrow p(0,0), 100000 > th$, and hence $\leftarrow hr(x)$ is satisfied in D^U. Thus, even if $100000 > th$, methods such as those mentioned above accept U, i.e., they fail to detect that U causes a high risk. Thus, their output is not reliable in the presence of extant low risks. As opposed to that, if $100000 > th$, then U is reliably rejected by each inconsistency-tolerant method, since the case $\leftarrow hr(0)$ is satisfied in D but violated in D^U. □

E. Unsatisfiability-tolerant Integrity Checking

By bad design or faulty schema updates, database evolution may lead to an *unsatisfiable* integrity theory, i.e., no state could ever satisfy integrity. Theoretically, unsatisfiable integrity is the worst possible situation, since each state then is irreparable. Since unsatisfiability is known to be undecidable in general, it even might never be detected. Anyway, with an unsatisfiable integrity theory, schema evolution may seem to have reached a dead end.

However, inconsistency-tolerant integrity checking can be applied even if the constraints are unsatisfiable, i.e., if integrity is inevitably violated in any state. By using an inconsistency-tolerant method, one can guarantee that all satisfied cases of constraints remain satisfied, even though integrity as a whole is never attainable. Thus, each inconsistency-tolerant method is also unsatisfiability-tolerant, as defined below.

***Definition** 5.4: An integrity checking method $\mathcal{M}$ is called* unsatisfiability-tolerant *if, for each database D, each unsatisfiable integrity theory IC and each update U,* (3) *holds.*

Definition 5.4 straightforwardly entails Corollary 15, since unsatisfiability of *IC* is in fact not excluded in Definition 3.2.

***Corollary** 15:* Each inconsistency-tolerant integrity checking method is unsatisfiability-tolerant.

***Example** 5.7:* Let *IC* be the unsatisfiable integrity theory $\{\leftarrow p(x,x), \leftarrow \sim p(0,0), \leftarrow q(x) \wedge r(x)\}$. Clearly, the first two denials in *IC* can never be satisfied at a time. However, in $D = \{p(0,0), q(0), r(0), q(1), r(2), q(3), r(4), q(5), \ldots, q(99), r(100)\}$, all basic cases of *IC* except $\leftarrow p(0,0)$ and $\leftarrow q(0) \wedge r(0)$ are satisfied. Although *IC* can never be fully satisfied, it makes sense to accept updates such as deleting $q(0)$, which would actually remove a case of violated integrity, and to prevent insertions, e.g., of $q(2)$, that would introduce new violations. Also, no inconsistency-tolerant method would ever reject any request to delete any fact from q or r. Or when, e.g., the insertion of a fact of the form $q(a)$ is requested, only the simplification $\leftarrow r(a)$ will be checked, i.e., the request is rejected only if $r(a)$ is in D^U. □

VI. Experimental evaluation

We now describe the experiments performed for evaluating, first, the benefits of ITIC for database updating, second, its benefits for query answering, and, third, the impact of ITIC on the performance of updating, checking, and querying. Each experiment is based on a series of updates, starting from an initial state and leading to a final state. Updates are either checked by ITIC, as proposed in this paper, or not checked at all, since checking updates in inconsistent databases is traditionally considered invalid. More precisely, we run the following three kinds of experiments, the setups of which are described in VI-A.

- Updates may change the amount of inconsistency. In VI-B, we assess how inconsistency varies between initial and final states, both when ITIC is used and when integrity is not checked. We do that by measuring the percentage of tuples that participate in constraint violations.
- Extant inconsistency may cause incorrect answers. In VI-C, we measure and compare the amounts of incorrect tuples in the answers to queries posed in the final states, both when ITIC is used and when integrity is not checked. That way, we obtain an indication of the quality of query answers depending on whether ITIC is used or not. We compute answers both by traditional query evaluation and by *consistent query answering* (CQA), a technique for improving the quality of query answers in the presence of inconsistency [1].
- Using ITIC obviously weighs in more on performance than running no integrity checks at all. In VI-D, we measure and report on the times required for integrity checking, updating and querying both when ITIC is used and when integrity is not checked.

A. Parameters and setups

The tests are run on the databases and queries of the TPC-H decision support benchmark[2], which is known to have a broad industry-wide relevance. In order to cover a significant spectrum of update series, we have experimented with the following variants of parameter values for the initial state and the updates.

- s: *initial state size*. We experiment with s = 100MB, 500MB, 1GB, 2GB. A database with s = 2GB has approximately 16 million tuples.
- p: *initial inconsistency*, expressed as the percentage of tuples that participate in constraint violations in the initial state. For simplicity, we only consider primary key constraint violations. We experiment with $p = 0\%, 1\%, 10\%, 33\%$. For example, $p = 10\%$ and s = 2GB means that 1,600,000 tuples

[2] http://www.tpc.org/tpch/

violate a primary key constraint. Violations occur when the same key value is repeated; we experiment with violations caused by 2 to 50 repetitions, and an equal percentage of violations in all tables.

- i: *percentage of insertions in the update series that violate a primary key constraint*. We experiment with i = 10%, 50%, 90%. We generate update series consisting of insertions and deletions of a size equal to 10% and, respectively, 1% of the size of the initial database, so as to simulate a significant evolution of the database. For example, with $i = 10\%$, and $s = 2$GB, there will be 160,000 insertions violating primary key constraints, i.e., 10% of 10% of 16 million tuples. (Note that deletions cannot cause any violation of primary key constraints.)

The TPC-H suite provides a script, called `dbgen`, for generating database states of a given size that satisfy the constraints. In order to set the initial inconsistency, we use the same technique as in [20], where the authors test their CQA approach against the TPC-H benchmark. To have, e.g., $p = 10\%$ and violations with 2 key repetitions each in a database of $s = 1$GB, `dbgen` is first used to create a consistent state of size 0.95GB; we denote that state by $\bar{D}_s^p$. Then, a set A of tuples of size 0.05GB from the database is randomly selected from a uniform distribution; a new set B of size 0.05GB is generated from this, with the same key values as in A, and non-key values taken randomly from the other tuples in the database. Then, set B is added to $\bar{D}_s^p$, and the resulting state D_s^p has the desired size s and inconsistency p. Similarly, we use `dbgen` to generate a set of updates $\mathcal{U}_s^i$ consisting of deletions (of size 1% of s) and insertions (10% of s), i percent of which introduce new constraint violations.

B. Measuring inconsistency variations through updates

The first measurement we have performed assesses the inconsistency, i.e., the percentage of tuples that violate a constraint, in the final state reached after a series of updates. We denote as D_{NC} (resp., D_{ITIC}) the state reached after executing on D_s^p the updates in $\mathcal{U}_s^i$ with no checking (resp., if accepted by an ITIC method). Both here and in the other tests, we use $\mathcal{M}_N$ as the integrity checking method, since it is sound, complete and inconsistency-tolerant. Figure 2 shows how inconsistency varies between the initial state D_s^p and the final states D_{NC} and D_{ITIC} for $s = 100$MB and for all possible values for p (lines with squares for $p = 0\%$, circles for $p = 1\%$, lozenges for $p = 10\%$, and triangles for $p = 33\%$) and i (light grey for $i = 10\%$, dark grey for $i = 50\%$, and black for $i = 90\%$). The dashed lines refer to the no-checking scenario, where inconsistency always grows, unless $i < p$ (which is the case in our tests only for $p = 33\%$ and $i = 10\%$). The continuous lines refer to the ITIC scenario, where the number of violations cannot increase; in fact, inconsistency naturally decreases, since the database tends to become bigger after executing our series of updates, while inconsistency does not increase. The differences in the amount of inconsistency between D_{NC} and D_{ITIC} are quite significant in many cases. For example, for $p = 1\%$ and $i = 10\%$, inconsistency amounts to 1.81% of the database in D_{NC}, while it is only 0.99% in D_{ITIC} (differences are even bigger for bigger values of i). Needless to say, if the initial state is consistent ($p = 0$), integrity is totally preserved with ITIC, while it is lost with no checking. Similar considerations hold also for the other values considered for s.

Another quantitative difference, not shown in Figure 2, between D_{NC} and D_{ITIC} regards their sizes. Obviously, D_{ITIC} does not contain any new violation and is therefore always smaller than D_{NC} whenever $i > 0$ (their difference increases as i increases). For example, for $p = 10\%$ and $i = 90\%$, the size varies from 953,088 tuples in D_{NC} to 875,109 in D_{ITIC}.

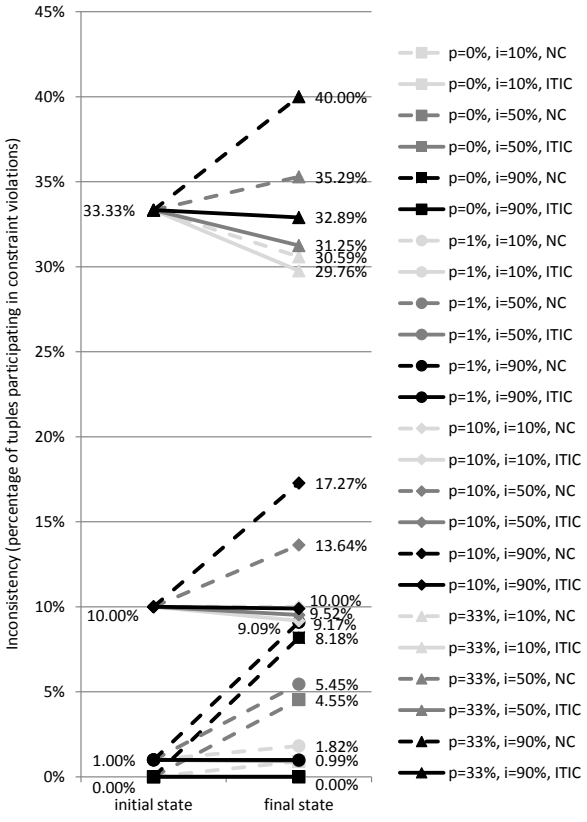

Fig. 2. Inconsistency measured after updates applied to an initial state with size $s = 100$MB and different values for p (initial inconsistency) and i (percentage of insertions in the update series violating a constraint), as indicated. Continuous lines indicate tests run with ITIC (where inconsistency never increases), while dashed lines indicate no checking (NC).

C. Measuring incorrect tuples in query answers

Our second experiment tests the negative effect of inconsistency on query answering, and to which extent such effect can be cured by handling database maintenance with ITIC.

Inconsistency may be responsible for incorrect answers to queries. Let us indicate with Q_D the set of tuples in the answer to query Q evaluated in database D. We define a tuple t to be a *correct* answer to Q in D if $t \in Q_{\bar{D}}$, where $\bar{D}$ is the *reference database* of D, i.e., the state in which D would be if no inconsistency had occurred at any time. Accordingly, we define the *false positives* of Q in D as the set $Q_D^+ = Q_D \setminus Q_{\bar{D}}$, the *false negatives* as $Q_D^- = Q_{\bar{D}} \setminus Q_D$. The smaller Q_D^+ and Q_D^-, the better the quality of Q_D. Determining the reference database for a given D requires, in general, information on all the updates that have led to D, which is typically unavailable. However, since in our experiments we have that kind of information, for our purposes it is sufficient to assume that $\bar{D}_s^p$ is the reference database of D_s^p.

Another way to remove inconsistency from D_s^p is to eliminate all tuples that participate in constraint violations. This is a very

strict repair procedure, since some of the eliminated tuples may indeed be correct. However, it is more feasible than determining the reference database, in general. We denote by $\tilde{D}_s^p$ the state obtained in this way. Because of its consistency, traditional integrity checking can be applied to any series of updates starting from $\tilde{D}_s^p$. We denote by D_{clean} the state obtained from $\tilde{D}_s^p$ by executing the updates in $\mathcal{U}_s^i$ accepted by $\mathcal{M}_N$.

Our tests measure and compare, for given benchmark queries, the amounts of false positives and negatives in D_{NC}, D_{ITIC}, and D_{clean}. To this end, consider that the database D_{ideal} obtained by executing on $\bar{D}_s^p$ the updates in $\mathcal{U}_s^i$ accepted by $\mathcal{M}_N$ is the reference database of each of D_{NC}, D_{ITIC}, and D_{clean}.

Queries that always return a fixed, very small number of results coming from complex aggregations are not well-suited for our purposes, since even tiny variations in the state may imply different aggregate values, and thus false positives and negatives. Therefore we choose to focus on queries that return at least 10 results, namely queries Q_3 and Q_{10} of the benchmark. Such queries are "top-k" queries that output the first few results of an aggregation operation.[3] Q_3 involves 3 relations, selects 4 attributes, and returns 10 results. Q_{10} involves 4 relations, selects 8 attributes, and returns 20 results.

In order to compare false positives and negatives of large query answers, we also consider queries Q_3^{all} and Q_{10}^{all}, that we define as identical to Q_3 and, resp., Q_{10}, but without being limited to the top 10 or, resp., 20 results.

Finally, we also consider the rewritings Q_3^{cqa} and Q_{10}^{cqa} of Q_3 and, resp., Q_{10} obtained by the CQA rewriting technique described in [20]. Intuitively, CQA consists in rewriting a given query Q over a database D with an integrity theory *IC* into a new, more complex query Q^{cqa}, the evaluation of which only produces the *consistent answers* to Q. In the definition of [1], [20], a tuple is a consistent answer to Q in D if it is an answer to Q in each consistent database whose set difference wrt. D is minimal. CQA can therefore be regarded as a technique for reducing the amount of incorrect answer tuples.

We measure $|Q_D^+|$ and $|Q_D^-|$ for every $Q \in \{Q_3, Q_{10}, Q_3^{all}, Q_{10}^{all}, Q_3^{cqa}, Q_{10}^{cqa}\}$ and every $D \in \{D_{\text{NC}}, D_{\text{ITIC}}, D_{\text{clean}}\}$. Note that $|Q_D^+| = |Q_D^-|$ for $Q \in \{Q_3, Q_{10}\}$, since the cardinality of the query answers is fixed by the "top-k" clause.

Figures 3 and 4 compare the amounts of false positives for Q_3 and, resp., Q_{10} in D_{NC}, D_{ITIC}, and D_{clean}, with $s = 100$MB and all combinations of p and i. The benefits of ITIC are significant, especially for lower amounts of initial inconsistency. For example, for $p = 1\%$ and $i = 10\%$, there are only 4 incorrect answers among the top 10 answers to Q_3 in D_{ITIC}, whereas all top 10 answers are incorrect in D_{NC}. The graph also signals that initial repairing is beneficial: removing all potentially incorrect data lowers false positives and negatives considerably. Recall, however, that repairing is costly.

Moreover, although the answers in D_{clean} are usually better than those in D_{ITIC}, it is not necessarily so, as shown, e.g., for Q_{10} with $i = 10\%$ and $p = 1\%$, where the number of false positives is 7 in D_{clean}, but 4 in D_{ITIC}. The other lines in the figure report the amounts of false positives for Q_3^{cqa} and Q_{10}^{cqa} in D_{NC} and D_{ITIC} (not in D_{clean}, since it is consistent, so the answers to Q_3^{cqa} and Q_{10}^{cqa} coincide with those to Q_3 and, resp., Q_{10}). Although slower in execution, such queries further improve the quality of answers, and in some cases they even eliminate all false positives in D_{ITIC}. This suggests that, for quality-critical OLTP applications, where some extra time is affordable for CQA but not for total repairs, ITIC should be used for database maintenance together with CQA for query answering. When the database is too inconsistent, as, e.g., for $p = 33\%$, an update phase of 10% the size of the database cannot do much to significantly improve consistency, so all top answers are incorrect both in D_{NC} and D_{ITIC} in Figure 4.

[3]The queries in the TPC-H specification are parameterized, and the standard suggests values for these parameters. In the experiments, we used the suggested values in all the queries.

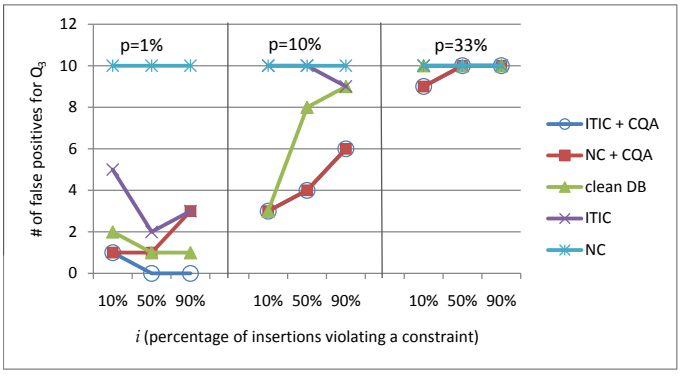

Fig. 3. The lines refer to an initial state size $s = 100$MB and different values for p (initial inconsistency) and i, and show the false positives for Q_3 on D_{NC} (stars), Q_3 on D_{ITIC} (crosses), Q_3 on D_{clean} (triangles), Q_3^{cqa} on D_{NC} (squares), and Q_3^{cqa} on D_{ITIC} (circles).

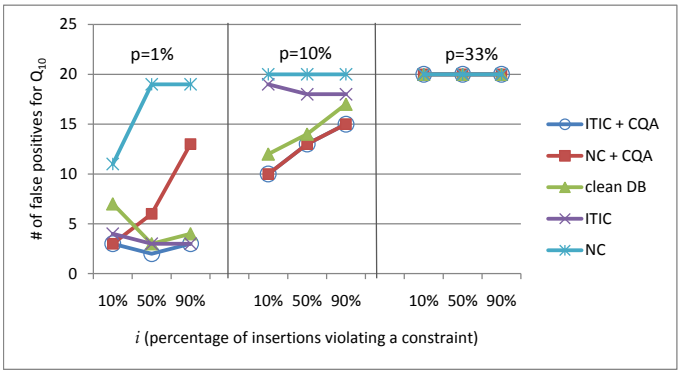

Fig. 4. The lines refer to an initial state size $s = 100$MB and different values for p (initial inconsistency) and i, and show the false positives for Q_{10} on D_{NC} (stars), Q_{10} on D_{ITIC} (crosses), Q_{10} on D_{clean} (triangles), Q_{10}^{cqa} on D_{NC} (squares), and Q_{10}^{cqa} on D_{ITIC} (circles).

Figure 5 shows the amounts of both false positives and negatives for Q_3^{all} and Q_{10}^{all}, for $s = 100$MB, $p = 1\%$, and for all values of i. Note that these values are higher than in Figures 3 and 4, since Q_3^{all} and Q_{10}^{all} do not restrict to the top 10, resp., 20 results. Again, the benefits of ITIC seem impressive.

Finally, Figure 6 shows the amount of false positives for Q_3^{all} and Q_{10}^{all} with $i = 50\%$ and $p = 1\%$ for different values of s, both in D_{NC} and in D_{ITIC}. We observe that the amount of false positives in D_{NC} is about 6 times higher than in D_{ITIC}, therefore with remarkable benefits due to ITIC. The mentioned factor depends of course on the chosen parameters and on the selection predicates in the queries, and can be explained as follows. Queries Q_3^{all} and Q_{10}^{all} turn out to retrieve a number of false positives that is proportional to the number of tuples violating the constraints, which, in turn, is proportional to s, since p is fixed. Since the

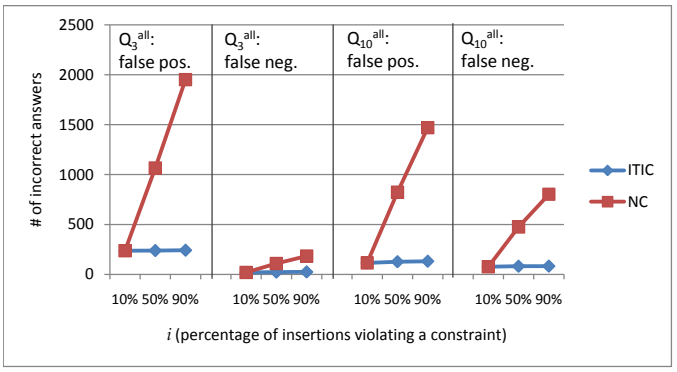

Fig. 5. False positives and negatives for Q_3^{all} and Q_{10}^{all} in D_{NC} (squares) and D_{ITIC} (diamonds) with initial state size $s = 100$MB, initial inconsistency $p = 1\%$, and different values for i.

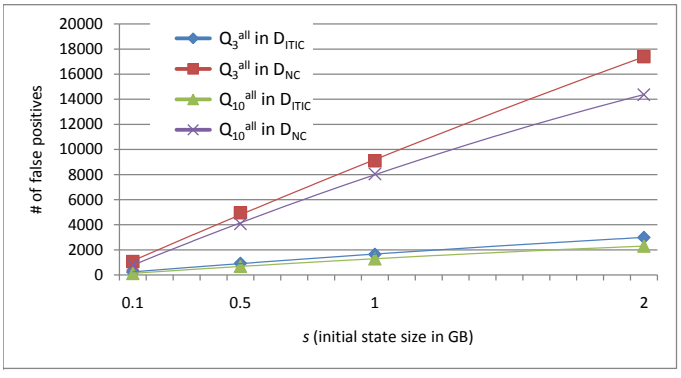

Fig. 6. False positives for Q_3^{all} in D_{NC} (squares) and D_{ITIC} (diamonds), and Q_{10}^{all} in D_{NC} (crosses) and D_{ITIC} (triangles) for $p = 1\%$ and $i = 50\%$ for different sizes of the initial state.

size of the insertions is $10\% \cdot s$, the inconsistency in D_{NC} is about $(i \cdot 10\% + p)/p = 6$ times higher than that in D_{ITIC}.

D. Measuring execution times

In our last test, we measure the time consumed for integrity checking, updating, and querying in the states obtained by all update series considered so far, i.e., leading 1) from D_s^p to D_{NC}, 2) from D_s^p to D_{ITIC}, 3) from $\tilde{D}_s^p$ to D_{clean}, 4) from $\bar{D}_s^p$ to D_{ideal}. The test machine sports a 2.66GHz Quad Core Intel processor with 4GB 800MHz RAM and runs Sqlserver 2005 under Windows XP.

Apart from 1), where no time is spent for integrity checking, there are little notable differences of measured times across different update series. For example, for $s = 1$GB and $p = 1\%$, integrity checking, if performed, always takes around 450 seconds in total, updating about 18 seconds, answering non-CQA queries 2.2 seconds, and answering CQA queries 12 seconds. For larger amounts of inconsistency, e.g., $p = 33\%$, improvements up to 10% of the execution times wrt. the update series 2) are observed both for query answering and for integrity checking in 4) and up to 20% in 3). These, however, are due to the different sizes of initial states, which are the smaller the higher the initial inconsistency: while D_s^p has size s, $\tilde{D}_s^p$ has size $(1-p) \cdot s$ and $\bar{D}_s^p$ has size $(1 - p/2) \cdot s$.

Note that we did not perform any particular tuning of the database, so as to speed up query answering or integrity checking.

E. Summary of experimental results

The experiments reported in this section have provided evidence of the following benefits of ITIC:

- A series of updates executed on a database state leads to lower amounts of inconsistency if filtered by ITIC, with no extra checking cost incurred with respect to integrity checking in a traditional, consistent setting.
- The query answers obtained from a database state reached after a series of updates checked by ITIC are generally better than without checking, in that they contain fewer false positives and negatives.
- Traditional query answering can be replaced by CQA to further improve the quality of query answers.

The smaller the initial amount of inconsistency and the larger the amount of inconsistency introduced by updates, the more the above effects become visible. Or, in other words, larger initial amounts of constraint violations require longer periods of updates checked by inconsistency-tolerant methods for decreasing the initial inconsistency significantly.

VII. Related Work

Various forms of exception handling for dealing with persistent inconsistencies as embodied by constraint violations have been proposed in [21], [22], [23] and others. However, integrity checking is not addressed in any of those works.

Another approach to deal with inconsistencies is to repair them (cf. V-B), which, despite recent advances [18], [17], is known to be intractable in general. Anyway, all approaches that either eliminate or work around inconsistencies (e.g., by repairing them or treating them as exceptions) need to know about extant integrity violations. As opposed to that, ITIC simply leaves inconsistencies alone. That works reliably, even if violated cases of constraints are unknown to the user or the application, as seen in V-D.

To the best of our knowledge, the putatively fundamental role alleged to total integrity as an indispensable premise for simplified integrity checking has never been challenged. That may be due to the classical *ex contradictione quodlibet* rule, by which conclusions derived from inconsistency cannot be considered reliable. However, in V-D, we have seen that, on the contrary, the use of inconsistency-tolerant methods is fully reliable.

On the other hand, it is astonishing that total integrity has always been insisted on, since many database contexts in practice suffer from some amount of inconsistency.

Nevertheless, interesting work has been going on in recent years under the banner of "inconsistency tolerance". A lot of it is concerned with consistent query answering in inconsistent databases (abbr. CQA) [1], [16], [19]. CQA defines answers to be correct if they are logical consequences of each reasonably repaired state of the database, i.e., each state that satisfies integrity and differs from the given violated state in some minimal way. CQA and ITIC have in common that they neither capitulate in the presence of inconsistency (as classical logic would), nor need to appeal to repairing violated constraints (as traditional query answering and integrity checking would). However their main purposes are different, since CQA enables query answering, while ITIC enables updating, even when the database violates integrity. Yet, integrity checking (which can be seen as a special-purpose variant of query answering) has, to the best of our knowledge, never been addressed in detail by the CQA community. As

was observed in Section VI, ITIC can be considered as largely complementary to CQA, since the former prevents new integrity violations to occur during updates but does not remove the effect that extant violations may have on query answers, which is precisely what the latter does.

Also, a variety of paraconsistent logic approaches that are tolerant and robust wrt. inconsistency have received some attention, e.g., [24], [25]. Most of them, however, deviate significantly from the syntax and semantics of classical first-order logic, while ITIC does not. Some paraconsistent approaches resort to modal or multivalued logic. As opposed to that, ours complies with conventional two-valued semantics of databases and integrity in the literature.

Yet, resolution-based query answering (by which each of the methods mentioned in this paper has been implemented) can be characterized as a procedural form of paraconsistent reasoning [26]. This is particularly noteworthy for proof procedures that use integrity constraints as candidate input clauses, such as those in [12], [4]. Thus, the paraconsistency of logic programming naturally qualifies it as a paradigm for implementing inconsistency-tolerant approaches to database integrity.

Further relevant work on the management of inconsistencies in databases comes from the field of inconsistency measuring [27].

Inconsistency measures are useful for updates and integrity checking if one wants to accept an update only if the measure of inconsistency of the old state does not increase in the new state. That, however, is precisely accomplished by ITIC, as soon as the set of violated cases of an integrity theory is measured: that set cannot be increased by an update if the update is checked by an inconsistency-tolerant method. Also other measures, such as those proposed in [27], should be useful for determining the increase or decrease of inconsistency across updates. Alternative ways to characterize ITIC, including definitions based on inconsistency measures, are described in [10]. There, different classes of integrity checking strategies are identified, studied and compared wrt. their inconsistency tolerance capabilities.

This paper improves and extends [28] in several ways. New are the properties and conditions for completeness and weak completeness wrt. inconsistency tolerance. Also the application of ITIC to various database management problems in section V is new. Another important addition of this paper is the validation of the practical relevance of ITIC in section VI.

VIII. Conclusion

The purpose of integrity checking is to ensure that the satisfaction of each constraint is preserved across updates. Traditionally, the theory of efficient integrity checking stipulates total integrity, i.e., that all constraints be satisfied in each state, without exception. In practice, however, that is an almost utopian wish.

To overcome this gap between theory and practice, we have relaxed the total integrity premise by a new requirement that tolerates inconsistency. Essentially, it asks that only those cases of constraints that are satisfied in the old state remain satisfied in the new state, while any amount of extant violated cases can be tolerated. (Cases are obtained from constraints by instantiating $\forall$-quantified variables that are not governed by $\exists$-quantified ones.)

We have seen that many (though not all) existing integrity checking methods comply with this relaxation without penalty, i.e., no change or adaptation of methods that can be shown to be inconsistency-tolerant is necessary at all. For such methods, traditional integrity checking becomes merely a special border case of our inconsistency-tolerant generalization.

The main benefits of inconsistency-tolerant integrity checking (ITIC) as identified in this paper can be summarized as follows.

1) The applicability of integrity checking methods is broadened significantly. ITIC allows updates to be fruitfully checked for integrity preservation even in the presence of inconsistency.

2) The application of ITIC tends to reduce the amount of inconsistency. In particular, ITIC guarantees that the number of violated basic cases cannot increase. Therefore, insertions cannot increase the percentage of inconsistency in the data either.

3) ITIC tends to improve the quality of answers to queries. Experimentally, we have shown that lower amounts of inconsistency obtained with ITIC typically result in lower amounts of false positives and negatives.

4) Procedural constructs for integrity maintenance can be avoided. Many applications do not comply with the demand of total integrity. Thus, instead of using methods for checking declarative constraints, application programmers often have resorted to less reliable procedural constructs, such as dynamic constraints, triggers or stored procedures. The results of this paper now legitimize the use of methods for ITIC, since their output is reliable also in the presence of inconsistency.

Future work includes further investigation of the interplay between the notion of inconsistency-tolerant repair, as introduced in V-B, and CQA. Instead of referring to total repairs for answering a query, as CQA does, it should be sufficient to be content with partial repairs that tolerate inconsistencies that do not "interfere" with the query. This would also mean that CQA could even deal with unsatisfiable theories without trivializing query answers (by definition, every n-tuple is in the CQA answer to an n-ary query if no repair exists).

Other pending work concerns inconsistency measures, as mentioned in Section VII. Acceptance of updates by an ITIC method depends on the measure in use, which, in this paper, is based on cases. Other measures may prove relevant for ITIC [10]. We also intend to investigate the capacity of inconsistency tolerance of abduction-based procedures, such as those described in [29]. Further ongoing studies are concerned with ITIC for concurrent transactions and replicated databases.

To conclude, we believe that the notion of ITIC can be embraced by producers and vendors of DBMSs at no additional cost in most of the existing implementations. Thus, the problematic use of triggers and other non-declarative constructs can be reduced in favor of ITIC, which is more useful and more reliable than methods that insist on total integrity.

Acknowledgment

Hendrik Decker has been supported by FEDER and the Spanish MEC grant TIN2006-14738-C02-01. Davide Martinenghi has been supported by the Search Computing (SeCo) project, funded by ERC under the 2008 Call for "IDEAS Advanced Grants". The authors also wish to thank Davide Barbieri for his valuable contribution to the experimental evaluation.

References

[1] M. Arenas, L. E. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases," in *PODS'99*. ACM Press, 1999, pp. 68–79.

[2] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill, 2003.

[3] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *Proc. of the 5th Logic Programming Symposium*, 1988, pp. 1070–1080.
[4] F. Sadri and R. Kowalski, "A theorem-proving approach to database integrity," in *Foundations of Deductive Databases and Logic Programming*. Los Altos, CA: Kaufmann, 1988, pp. 313–362.
[5] J. W. Lloyd, L. Sonenberg, and R. W. Topor, "Integrity constraint checking in stratified databases," *JLP*, vol. 4, no. 4, pp. 331–343, 1987.
[6] J.-M. Nicolas, "Logic for improving integrity checking in relational data bases," *Acta Informatica*, vol. 18, pp. 227–253, 1982.
[7] H. Christiansen and D. Martinenghi, "On simplification of database integrity constraints," *Fundamenta Informaticae*, vol. 71, no. 4, pp. 371–417, 2006.
[8] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom, "Constraint checking with partial information," in *PODS'94*. ACM Press, 1994, pp. 45–55.
[9] S. Y. Lee and T. W. Ling, "Further improvements on integrity constraint checking for stratifiable deductive databases," in *VLDB'96*. Morgan Kaufmann, 1996, pp. 495–505.
[10] H. Decker and D. Martinenghi, "Classifying integrity checking methods with regard to inconsistency tolerance," in *PPDP'08*. ACM Press, 2008, pp. 195–204.
[11] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational data bases," 1977, pp. 77–90.
[12] R. A. Kowalski, F. Sadri, and P. Soper, "Integrity checking in deductive databases," in *VLDB'87*. Morgan Kaufmann, 1987, pp. 61–69.
[13] A. Guessoum and J. Lloyd, "Updating knowledge bases," *New Generation Computing*, vol. 8, no. 1, pp. 71–89, 1990.
[14] ——, "Updating knowledge bases ii," *New Generation Computing*, vol. 10, no. 1, pp. 73–100, 1991.
[15] A. Kakas and P. Mancarella, "Database updates through abduction," in *VLDB'90*. Morgan Kaufmann, 1990, pp. 650–661.
[16] G. Greco, S. Greco, and E. Zumpano, "A logical framework for querying and repairing inconsistent databases," *IEEE TKDE*, vol. 15, no. 6, pp. 1389–1408, 2003.
[17] T. Eiter, M. Fink, G. Greco, and D. Lembo, "Repair localization for query answering from inconsistent databases," *ACM TODS*, vol. 33, no. 2, 2008.
[18] F. Furfaro, S. Greco, and C. Molinaro, "A three-valued semantics for querying and repairing inconsistent databases," *Ann. Math. Artif. Intell.*, vol. 51, no. 2-4, pp. 167–193, 2007.
[19] J. Chomicki, "Consistent query answering: Five easy pieces," in *ICDT*, 2007, pp. 1–17.
[20] A. Fuxman, E. Fazli, and R. J. Miller, "ConQuer: Efficient management of inconsistent databases," in *SIGMOD'05*, 2005, pp. 155–166.
[21] A. Borgida, "Language features for flexible handling of exceptions in information systems," *ACM TODS*, vol. 10, no. 4, pp. 565–603, 1985.
[22] R. Balzer, "Tolerating inconsistency," in *ICSE'91*. IEEE Computer Society / ACM Press, 1991, pp. 158–165.
[23] O. Etzion, "Handling active databases with partial inconsistencies," in *ECSQARU'91*, vol. 548. Springer LNCS, 1991, pp. 171–175.
[24] L. Bertossi, A. Hunter, and T. Schaub, *Inconsistency Tolerance*, ser. LNCS. Springer, 2005, vol. 3300.
[25] J. Grant and A. Hunter, "Analysing inconsistent first-order knowledgebases," *Artif. Intell.*, vol. 172, no. 8-9, pp. 1064–1093, 2008.
[26] R. A. Kowalski, *Logic for Problem Solving*. Elsevier, 1979.
[27] J. Grant and A. Hunter, "Measuring inconsistency in knowledgebases." *J. of Intelligent Information Systems*, vol. 27, no. 2, pp. 159–184, 2006.
[28] H. Decker and D. Martinenghi, "A relaxed approach to integrity and inconsistency in databases," in *LPAR'06*, vol. 4246. Springer, 2006, pp. 287–301.
[29] P. M. Dung, R. Kowalski, and F. Toni, "Dialectic proof procedures for assumption-based admissible argumentation," *Artif. Intell.*, vol. 170, no. 2, pp. 114–159, 2006.

Hendrik Decker Dr. Decker graduated in computer science at the Technical University of Munich, Germany, in 1978. He received the degree of doctor of engineering sciences at the University of Kaiserslautern, Germany, in 1982. He then worked as a researcher, developer and consultant in several labs associated to Siemens Corporate R&D, among them the Knowledge Base group of the European Computer-Industry Research Centre in Munich (1984–1990). Since 2002, he is a researcher in residence at the Instituto Tecnológico de Informática in Valencia, Spain. His main area of interest has always been computational logic, and in particular the semantic integrity and consistency of stored data.

Davide Martinenghi Dr. Martinenghi received a MS as Computer Engineer from Politecnico di Milano, Italy, in 1998 and a Ph.D. in Computer Science from Roskilde University, Denmark, in 2005 with a dissertation on integrity checking for deductive databases. Presently, he is assistant professor at Politecnico di Milano. His main interests are data integrity maintenance, data integration, logic programming, knowledge representation, and, in a broad sense, applications of logic to databases. He is currently focusing on inconsistency tolerance in database systems as well as on query optimization aspects related to web data access and web search.