A Measured Evolution of Database Integrity

Hendrik Decker[⋆]

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Spain

Abstract. Inconsistency in large database systems is commonplace and therefore must be controlled in order to not get out of hand. Consistency in database systems is encoded by integrity constraints. Inconsistency thus corresponds to constraint violations. Database system services need to function in spite of extant integrity violations, but inconsistency should not increase beyond control in the course of the evolution of such systems. Evolution is effected by updates that may involve insertions and deletions of relational facts as well as schema updates. We show how to determine the causes of violations. Knowledge about such causes can be used to control inconsistency: an increase of integrity violations by updates can be prevented, while tolerating extant inconsistencies, even if the database schema is altered, and even if the schema is unsatisfiable.

1 Introduction

Whenever a database or its schema is updated, its integrity should be checked. In case the update would cause the violation of an integrity constraint, it should either be rejected, or a warning should be issued such that the user or the application may modify the update, so that consistency will be preserved.

However, integrity is not always checked, be it out of neglect, or for trading off consistency for performance, or due to the incompatibility of legacy data with constraints, or for other reasons. Hence, inconsistency tolerance is needed, while an increase of integrity violations by updates should be avoided. Unfortunately, conventional integrity checking methods are not meant to be applied in the presence of inconsistency. They all require that integrity be totally satisfied in the state before the update, in order to check efficiently if integrity remains satisfied in the updated state.

Example 1. Consider the two constraints $I = \leftarrow married(x, y),\ male(x),\ male(y)$ and $I' = \leftarrow married(x, y),\ female(x),\ female(y)$ in a database system D of a civil registry office. They deny cosexual marriages (which is technically more convenient than stating and processing that the partners of each married couple must be of opposite gender). Yet, suppose D contains the fact *married*(*jay*, *pat*), where *jay* and *pat* are both male, perhaps due to a gender reversal after nuptials. Clearly, that amounts to a violation of I. Thus, conventional integrity checking methods are no longer applicable for updates of D, since they require that D does not violate integrity before checking any update for integrity preservation.

⋆ Supported by ERDF/FEDER and the Spanish MEC grant TIN2012-37719-C03-01.

Obviously, that is unnecessarily restrictive, in general. For instance, consider the update $U = insert\ married(joe, susan)$, which should be acceptable, because it is independent of the extant violation of I and does not cause any new violation.

Similarly, also schema updates, such as, e.g., inserting the constraint $I'' = \leftarrow married(x, y), married(x, z), y \neq z$, which forbids bigamy, is traditionally not considered to be efficiently checkable in the presence of extant constraint violations, even if I'' is not violated by any tuple in D.

In [15,16], we have shown that, contrary to common belief, most, though not all integrity checking methods are inconsistency-tolerant, i.e., the total satisfaction requirement can simply be waived without incurring any penalty. In other words, there is no problem with checking updates such as U in Example 1 in the presence of inconsistency, provided the used method is inconsistency-tolerant.

More precisely, we show in [15,16] that each inconsistency-tolerant method guarantees the preservation of the satisfaction of all ‘cases’ (instances of constraints) that were satisfied in the old state, even if other instances are violated. Hence, such methods make sure that inconsistency never increases across checked updates of base facts and view definitions, while tolerating extant inconsistencies.

In [6], we have shown that case-based inconsistency tolerance is possible also for schema updates. There, we have also seen that integrity violation can be controlled by inconsistency-tolerant methods if new or modified constraints are checked only against current and future updates, but not against legacy data. Moreover, we have shown in [6] that inconsistency tolerance can guarantee a reliable handling of hard and soft constraints through schema updates.

In [8], we have shown that, for *definite* databases and constraints, essentially the same guarantees of integrity preservation can be made if, instead of *cases* of constraint violations, the increase of *causes* of violations across updates of base facts and view definitions is checked. Causes as defined in [8] are minimal sets of database clauses, the presence or absence of which is responsible for integrity violations. As opposed to cases, causes also provide a basis for computing answers that have integrity in spite of extant violations, as shown in [9].

In [10], we have shown that cause-based inconsistency tolerance is also possible for updates of relational databases that involve schema updates, and that all advantages obtained by case-based inconsistency-tolerant methods for schema updates also are achieved by cause-based methods. However, the minimality condition for causes as defined in [8,10] does not scale up to non-monotonic negation. For instance, $\emptyset$ but not $\sim q$ would be a cause for the answer *no* to the query $\leftarrow p$ in $D = \{p \leftarrow q\}$.

In this paper, we are going to see that the generalization of causes in [9] to justify the integrity of positive and negative answers in definite and non-definite databases works as well also for controlling the evolution of integrity, i.e., for updates including schema alterations. Moreover, the generalized definition in [9] is formalized in a much simpler and less circumstantial manner in this paper.

2 Preliminaries

Our terminology and notation largely adhere to *datalog* [1]. In particular, for each database D, the well-known completion [4] of D is denoted by $comp(D)$, and $\models$ denotes logical consequence, i.e., truth in all Herbrand models. We assume a universal underlying language $\mathcal{L}$, of which $\mathcal{L}^c$ be the set of constant terms in $\mathcal{L}$ and $\mathcal{H}_{\mathcal{L}}$ the Herbrand base of $\mathcal{L}$.

In 2.1, we recapitulate some database fundamentals. In 2.2, we characterize the updates that we are going to deal with. In 2.3, we abstractly define the concept of integrity checking as implemented by conventional integrity checking methods. Later, in 3.3, that definition will be generalized in terms of inconsistency tolerance, based on a concept of causes, to be introduced in Section 3.

2.1 Basics

Each *database schema* (in short, *schema*) consists of a set T of *table definitions*, a set V of *view definitions* and a set IC of *integrity constraints*, a.k.a. *integrity theory*. Each table in T uniquely corresponds to a *base predicate* with some arity. Each view in V corresponds to a *view predicate*, defined by a set of clauses of the form $A \leftarrow B$ where A is an atom and B a conjunction of literals with predicates that recur on base predicates. Each constraint in IC is a first-order predicate logic sentence. Unless explicitly stated otherwise, each constraint be represented in the form of a *denial*, i.e., a clause without head whose body is a conjunction of literals.

A *database* is a pair $D = (S, E)$ where S is a schema and E is an *extension* of the table definitions in S, i.e., a set of ground base predicate facts. As usual, we assume that, for each schema $S = (T, V, IC)$ and each database $D = (S, E)$, $V \cup E$ determines a unique minimal Herbrand model, sometimes called the *standard model* of D.

For a database D, an integrity theory IC and $I \in IC$, let $D(IC) = sat$ (resp., $D(I) = sat$) denote that IC (resp., I) is satisfied in D, and $D(IC) = vio$ (resp., $D(I) = vio$) that it is violated. ‘Consistency’ and ‘inconsistency’ are synonymous to ‘satisfied’ and, resp., ‘violated’ integrity.

From now on, let D always denote a database, IC the integrity theory of the schema of D, I a constraint in IC, E the extension of D, and U an update.

We may use ‘;’ as a delimiter between elements of sets, for avoiding confusion with the use of ‘,’ which also symbolizes conjunction of literals in the body of clauses.

2.2 Updates

Updates map databases to databases. For a database D with view definitions V and an update U, let D^U denote the database into which D is mapped by U, V^U the views of D^U and IC^U the integrity theory of D^U. D is also called the *old* database and D^U the *new* database. If $IC^U = IC$, U is called a *conventional update*. Let U_c denote the maximal subset of U that is a conventional update. Moreover, if $V^U = V$ and $IC^U = IC$, U is called an *extensional update*.

2.3 Integrity Checking

From now on, let $\mathcal{M}$ always denote an integrity checking method (in short, *method*). Each $\mathcal{M}$ can be formalized as a function that maps pairs (D, U) to $\{ok, ko\}$, where *ok* means that $\mathcal{M}$ sanctions U and *ko* that it does not. The computation of $\mathcal{M}(D, U)$ usually involves less access to E, and thus is more efficient than a *brute-force* check, i.e., a plain evaluation of all constraints against the entire database, without any simplification.

For simplicity, we only consider methods $\mathcal{M}$ and classes of pairs (D, U) such that the computation of $\mathcal{M}(D, U)$ terminates. That can always be achieved by a timeout mechanism with output *ko*.

Below, Definition 1 captures the conventional concept of soundness and completeness of integrity checking methods for updates including schema updates.

Definition 1. (*Conventional Integrity Checking*)
$\mathcal{M}$ is called *sound* or, resp., *complete* if, for each (D, U) such that $D(IC) = sat$, the implication (1) or, resp., (2) holds.

$$\mathcal{M}(D, U) = ok \;\;\Rightarrow\;\; D^U(IC^U) = sat \tag{1}$$

$$D^U(IC^U) = sat \;\;\Rightarrow\;\; \mathcal{M}(D, U) = ok \tag{2}$$

Both (1) and (2) relate the output *ok* of $\mathcal{M}$ to integrity satisfaction. We omit symmetric relationships of *ko* and integrity violation, since soundness and, resp., completeness for *ko* and violation is equivalent to (2) and, resp., (1).

Example 2.
Let $IC = \{I, I'\}$ and U be as in Example 1, and suppose that $D(IC) = sat$. Most methods $\mathcal{M}$ evaluate the simplified constraints $\leftarrow male(joe), male(susan)$ and $\leftarrow female(joe), female(susan)$, and output $\mathcal{M}(D, U) = ok$ if *joe* is registered as male and *susan* as female (and, curiously, also if *joe* would be registered as female and *susan* as male). Otherwise, $\mathcal{M}(D, U) = ko$.

3 Causes

In this section, we first revisit the definitions of causes for integrity violation in definite databases [8] and for constraints with negation in relational databases [10]. Then, we extend them to non-definite databases and constraints by simplifying a previous generalization in [9].

Informally speaking, causes are minimal explanations of why an answer is given or why a constraint is violated. In 3.1, we formalize this idea in a homogeneous form for definite databases, queries and constraints, as well as for constraints with negation in relational databases. In 3.2, we generalize the preceding definitions for 'normal' deductive databases, queries and constraints with non-monotonic negation in the body of clauses, as well as for negative answers. In 3.3, we outline the use of causes for inconsistency-tolerant integrity checking, which is the cornerstone for an inconsistency-tolerant control of the evolution of databases, and particularly of schema updating, as addressed in Section 4.

3.1 Simple Causes

For the definition below, suppose that D is definite. Also, recall that the violation of a constraint of the form $\leftarrow B$ corresponds to a non-empty answer to the query $\leftarrow B$.

Definition 2. (*Causes for Definite Databases, Queries and Constraints*)
Let V be the view definitions and E the extension of a database D, V^* the set of all ground instances of V, B a conjunction of atoms and θ an answer to $\leftarrow B$ in D. A subset C of $V^* \cup E$ is called a *cause* of θ, and also a *cause* of $\forall(B\theta)$ in D, if $C \models \forall(B\theta)$ and there is no proper subset C' of C with that property. If $\leftarrow B$ is a constraint, we also call C a *cause of the violation* of $\leftarrow B$ in D.

Example 3.
Let D be as in Example 1. Clearly, $\{married(jay, pat),\ male(jay),\ male(pat)\}$ is a cause of the violation of I in D.

Causes according to Definition 2 can be computed straightforwardly, and essentially come for free, by tracing SLD refutations of denials [9]. However, due to the non-monotonicity of database negation, both the definition and the computation of causes become much more involved for causes of negative answers, and for causes of answers and constraint violations in databases with clauses that may contain negative literals in their body.

Small first steps into that direction had been taken in Section 6 of [8] and [10], for negative answers as well as queries and constraints with negative literals, but only for flat relational databases. A variant of the corresponding definition which likens it more to Definition 2 is reproduced below.

Definition 3. (*Causes for Negation in Relational Databases*)
Let D be a relational database, B a conjunction of literals and θ an answer to $\leftarrow B$ in D. A set C of ground literals such that $comp(D) \models C$ is called a *cause* of θ, and also a *cause* of $\forall(B\theta)$ in D, if $C \models \forall(B\theta)$ and there is no proper subset C' of C with that property. If $\leftarrow B$ is a constraint, we also call C a *cause of the violation* of $\leftarrow B$ in D.

Example 4. Let $I^{\sim} = \leftarrow married(x,y), male(x), {\sim}female(y)$ and D as in Example 1. Clearly, a cause of the violation of $I^{\sim}$ in D is $\{married(jay, pat),\ male(jay),\ {\sim}female(pat)\}$.

In fact, the definition in [8] is somewhat more general than Definition 3, in that $\leftarrow B$ is replaced by an arbitrary first-order sentence. Yet, for queries, both versions of the definition are fairly bland, since each answer in a flat relational database essentially explains itself. Thus, the computation of causes in relational databases is even simpler than the computation of causes in the definite case. However, Definition 3 is as interesting as Definition 2 for explaining the violation of constraints, since the facts that cause violations can be used for explanations [11] and repairs [12].

3.2 General Causes

Due to the non-monotonicity of database negation, the definitions in 3.1 do not generalize to explanations of negative answers, nor of answers and constraint violations for *normal* databases, queries and integrity theories, where negative literals may occur in the body of clauses. An extension of the definition of causes in [8] in order to cope with database negation had first been proposed in [9]. Below, we present a less complicated but equivalent version of that extension.

First, we recall that *comp*(D) essentially consists of the if-and-only-if completions (in short, completions) of all predicates in $\mathcal{L}$. For a predicate p in $\mathcal{L}$, let p_D denote the completion of p in D.

Definition 4. Let D be a database, p a predicate in $\mathcal{L}$, n the arity of p, $x_1, \ldots, x_n$ the $\forall$-quantified variables in p_D and θ a substitution of $x_1, \ldots, x_n$. For $A = p(x_1, \ldots, x_n)\theta$, the *completion* of A in D is obtained by applying θ to p_D and is denoted by A_D. Further, let $\underline{\mathit{comp}}(D) = \{A_D \mid A \in \mathcal{H}_{\mathcal{L}}\}$, and *if*$(D)$ and *only-if*(D) be obtained by replacing $\leftrightarrow$ in each $A_D \in \underline{\mathit{comp}}(D)$ by $\leftarrow$ and, resp., $\rightarrow$. Finally, let *iff*(D) = *if*$(D) \cup$ *only-if*(D). The usual equality axioms of *comp*(D) that interpret $=$ as identity be associated by default also to *iff*(D).

Clearly, *if*(D) is equivalent to the set of all ground instances of clauses in D. Moreover, *comp*(D), $\underline{\mathit{comp}}(D)$ and *iff*(D) clearly have the same logical consequences. However, the characterization of causes in Definition 5 below by minimal subsets of *iff*(D) is more precise than it could be if subsets of *comp*(D) were used instead.

Definition 5. (*Causes for Normal Databases, Queries and Constraints*)
Let D be a normal database, B a conjunction of literals and θ an answer to $\leftarrow B$ in D. A subset C of *iff*(D) is called a *cause* of θ, and also a *cause* of $\forall(B\theta)$ in D, if $C \models \forall(B\theta)$ and there is no proper subset C' of C with that property. If $\leftarrow B$ is a constraint, we also call C a *cause of the violation* of $\leftarrow B$ in D.

For easy reading, we may represent elements of *only-if*(D) in a simplified form, in subsequent examples. Simplifications are obtained by replacing ground equations with their truth values and by common equivalence-preserving rewritings for the composition of subformulas with *true* or *false*. Also, we may identify D with (V, E), while representing constraints or integrity theories explicitly, and omitting T.

Example 5.

a) Let $D = \{p(x) \leftarrow q(x), r(x);\ q(1);\ q(2);\ r(2);\ s(1);\ s(2)\}$. The only cause of the violation of $\leftarrow s(x), {\sim}p(x)$ in D is $\{s(1);\ p(1) \rightarrow q(1) \wedge r(1);\ {\sim}r(1)\}$.

b) Let $D = \{p \leftarrow q(1, x);\ q(2, y) \leftarrow r(y);\ r(1)\}$. The only cause of the violation of $\leftarrow {\sim}p$ in D is $\{p \rightarrow \exists x\, q(1, x)\} \cup \{{\sim}q(1, i) \mid i \in \mathcal{L}^c\}$.

c) Let $D = \{p \leftarrow q(x, x);\ q(x, y) \leftarrow r(x), s(y);\ r(1);\ s(2)\}$. Each cause of ${\sim}p$ in D contains $\{p \rightarrow \exists x\, q(x, x)\} \cup \{q(i, i) \rightarrow r(i) \wedge s(i)) \mid i \in \mathcal{L}^c\} \cup \{{\sim}r(2);\ {\sim}s(1)\}$ and, for each $j > 2$ in $\mathcal{L}^c$, either ${\sim}r(j)$ or ${\sim}s(j)$, and nothing else.

d) Let $D = \{p \leftarrow q;\ p \leftarrow {\sim}q\}$, $D' = D \cup \{q\}$ and $I = \leftarrow p$. Clearly, D is a cause of the violation of I in D and in D'. Another cause of p in D is $\{p \leftarrow {\sim}q;\ {\sim}q\}$. Another cause of p in D' is $\{p \leftarrow q;\ q\}$.

e) Let $D = \{p \leftarrow {\sim}q;\ q \leftarrow {\sim}r;\ r \leftarrow {\sim}s;\ r \leftarrow {\sim}t\}$. The two causes of the violation of $\leftarrow p$ in D are $\{p \leftarrow {\sim}q;\ q \rightarrow {\sim}r;\ r \leftarrow {\sim}s;\ {\sim}s\}$ and $\{p \leftarrow {\sim}q;\ q \rightarrow {\sim}r;\ r \leftarrow {\sim}t;\ {\sim}t\}$.

f) Let $D = \{p(x) \leftarrow r(x);\ r(1)\}$ and $I = \exists x(r(x) \wedge {\sim}p(x))$. A denial form of I is $\leftarrow \mathit{vio}$, where vio is defined by $\{\mathit{vio} \leftarrow {\sim}q;\ q \leftarrow r(x), {\sim}p(x)\}$, where q is a fresh 0-ary predicate. Thus, the causes of the violation of I in D are the causes of vio in $D' = D \cup \{\mathit{vio} \leftarrow {\sim}q;\ q \leftarrow r(x), {\sim}p(x)\}$. Thus, for each $\mathcal{K} \subseteq \mathcal{L}^c$ such that $1 \in \mathcal{K}$, $\{\mathit{vio} \leftarrow {\sim}q\} \cup \{p(i) \leftarrow r(i) \mid i \in \mathcal{K}\} \cup \{q \rightarrow \exists x(r(x) \wedge {\sim}p(x))\} \cup \{{\sim}r(i) \mid i \notin \mathcal{K}\}$ is a cause of vio in D'.

3.3 Cause-Based Inconsistency Tolerance

By common sense, an update U should be rejected only if U would cause the violation of a constraint, independent of the presence or absence of violations that are independent of U. However, conventional methods require that all constraints be satisfied before any update could be checked for integrity preservation.

Example 6. As in Example 1, let us assume a database D with constraints for preventing bigamy, married minors and other unlawful relationships. Yet, D may contain entries of persons married twice, underage spouses, etc. Such integrity violations may be due to omissions (e.g., an unrecorded divorce), neglect (e.g., integrity checking switched off for schema evolution) or other irregularities (e.g., a changed marriage legislation). Although new marriages that satisfy all constraints can be entered without problems into D, such updates have traditionally been considered to be not checkable by conventional methods if D contains facts that violate integrity.

As opposed to conventional approaches, cause-based integrity checking, as defined below, is inconsistency-tolerant, since it may sanction updates that do not increase the amount of causes of integrity violation, no matter how high the amount of inconsistency is before the update is executed. As we are going to see in Example 11, Definition 6 entails that even each unsatisfiable schema is tolerable, for updates that do not violate any constraint.

Definition 6. (*Cause-based Integrity Checking*)
a) Let $\mathit{cv}(D, I) = \{C \mid C$ is a cause of violation of I in $D\}$ denote the set of all causes of the violation of I in D, *modulo renamings of variables.*

b) A method $\mathcal{M}$ is called *sound* and, resp., *complete* for cause-based inconsistency-tolerant integrity checking if, for each tuple (D, U), implication 3 or, resp., 4 holds.

$$\mathcal{M}(D, U) = \mathit{ok} \ \Rightarrow\ \mathit{cv}(D^U, I) \subseteq \mathit{cv}(D, I) \text{ for each } I \in \mathit{IC} \tag{3}$$

$$\mathit{cv}(D^U, I) \subseteq \mathit{cv}(D, I) \text{ for each } I \in \mathit{IC} \ \Rightarrow\ \mathcal{M}(D, U) = \mathit{ok} \tag{4}$$

Definition 6 is going to be illustrated by examples 7 and, later, 8–12. Examples 7 and 12 feature extensional updates. Examples 8–10 are schema updates involving changes of tables, view definitions or integrity constraints. Example 12 also features the method in [17], which is not inconsistency-tolerant.

Example 7. For a table p, $I = \leftarrow p(x, y), p(x, z), y \neq z$ is a primary key constraint on the first column of p. Updates such as $U = \mathit{insert}\; p(a, b)$ can be accepted by each cause-based inconsistency-tolerant method $\mathcal{M}$, unless another p-tuple with the same key, e.g., $p(a, c)$, is in the database. In that case, $\{p(a, b), p(a, c)\}$ would be a new cause that violates I, and $\mathcal{M}$ would have to reject U. Otherwise, $\mathcal{M}$ can accept U, independent of any extant violation of I, e.g., by stored facts $p(b, b)$ and $p(b, c)$. Then, $\{p(b, b),\ p(b, c)\}$ is a cause of the violation of I, but no new cause of the violation of I is introduced by U. Since U does not increase the amount of causes of inconsistency, U can be accepted.

In [16], *case-based* inconsistency-tolerant integrity checking is studied. Case-based methods make sure that the amount of violated cases of constraint does not increase across updates. In [15,16], we have shown that many, though not all methods in the literature are sound for case-based inconsistency tolerance. An analogous result for cause-based methods is

Theorem 1. Each method shown to be sound for case-based integrity checking in [15,16] also is sound for cause-based inconsistency-tolerant integrity checking of conventional updates.

Proof. Theorem 1 follows from the more general result that each case-based method whatsoever is cause-based for conventional updates and updates of view definitions. That has been shown for definite databases and denials in [8], but that proof generalizes straightforwardly to normal databases and arbitrary integrity constraints.

In the remainder of this paper, we are going to show that the results for checking arbitrary schema updates with case-based methods in [6] continue to hold also for cause-based methods.

4 Inconsistency-Tolerant Schema Updating

In 4.1 we are going to show that cause-based methods that are inconsistency-tolerant for checking conventional updates in normal databases with normal constraints can also be used for schema updates involving changes in the integrity theory. We illustrate that result for table alterations in 4.2, for updates of integrity theories in 4.3, and for updates of view definitions in 4.4. In 4.3, we also propose an inconsistency-tolerant checking policy that makes schema updates more efficient. In 4.4, we also show that cause-based integrity checking even tolerates unsatisfiable schema definitions. In 4.5, we show that cause-based inconsistency tolerance is applicable to control the preservation of integrity for safety-critical schema updates in evolving databases.

4.1 Cause-Based Checking of Schema Updates

Most conventional integrity checking methods are not conceived for insertions or alterations of integrity constraints. (No checking is needed for the deletion of constraints since that may never cause any integrity violation.) However, to check new or altered constraints cannot take advantage of the incrementality of checking updates of base facts or view definitions. Rather, new or altered constraints usually are evaluated brute-force against the updated state, in order to check whether they are violated or not.

The following result states that methods that are cause-based for conventional updates continue to be inconsistency-tolerant according to Definition 6 for arbitrary schema updates, involving table alterations, updates of view definitions and integrity theories.

Theorem 2.
Each method $\mathcal{M}$ that is sound for cause-based integrity checking of conventional updates also is sound for cause-based integrity checking of schema updates.

Proof. For updates that involve only base facts and view definitions, the result follows from the generalization of the result in [8], as mentioned in the proof of Theorem 1. For updates involving changes in the integrity theory, the result can be shown by verifying (3) and (4) of Definition 6 by induction on the number of constraints inserted by U, assuming that each $I \in IC^U \setminus IC$ is evaluated brute-force in D^U.

Theorem 2 addresses the *soundness* of cause-based methods for schema updates. With regard to scaling up the *completeness* of cause-based methods from conventional updates to schema updates including modifications of the integrity theory, a somewhat surprising result is going to be presented in 4.3.

In the following subsections, we illustrate Theorem 2 by examples of checking various kinds of schema updates with cause-based methods: table alterations in 4.2, modifications of integrity theories in 4.3, alterations of view definitions and updates of databases with an unsatisfiable schema in 4.4, and updates by safety-critical applications in 4.5.

4.2 Inconsistency-Tolerant Table Alterations

Example 8 is going to illustrate that the amount of causes of violation never increases whenever any table alteration is checked by an inconsistency-tolerant method. It may even decrease, since some causes of violation may disappear by altering tables. Thus, cause-based inconsistency-tolerant methods support *inconsistency-tolerant partial repairs* [12], i.e., updates that reduce the amount of violated causes, while surviving violations are tolerated.

Example 8. Let $I = \leftarrow q(x, x, y)$ constrain q to be void of tuples the first and second arguments of which coincide. Let $q(a, a, a)$, $q(a, a, b)$ be the only facts in D that match $q(x,x,y)$. Clearly, each of them is a cause of the violation of I. Hence, conventional integrity checking refuses to handle the update U which alters the definition of q by swapping the second and third columns. As opposed to that, each complete cause-based inconsistency-tolerant method $\mathcal{M}$ will admit U and output *ok* if D contains no fact matching $q(x,y,x)$ such that $x \neq a$, and *ko* otherwise (e.g., if $q(b,a,b) \in D$). Thus, a sanctioned U does not only contain, but even diminish $cv(D,I)$.

4.3 Inconsistency-Tolerant Integrity Updates

As seen in 4.1, inconsistency tolerance of each cause-based method $\mathcal{M}$ for arbitrary schema updates U in databases D can be achieved by computing $\mathcal{M}(D,\, U_c)$ and evaluating each inserted or modified constraint brute-force against D^U. (For deleted constraints, no checking is needed. For example, to delete I and I' in Example 1 clearly cannot cause any integrity violation.)

Example 9. Assume U and $IC = \{I\}$ as in Example 7, and let U' consist of U and the insertion of the constraint $I' = \leftarrow p(x,x)$. Further, let us assume that U does not cause any violation of I. Thus, a brute-force evaluation of I' against D^U will result in $\mathcal{M}(D,U) = ok$ if $D(I') = sat$, and $\mathcal{M}(D,U) = ko$ if not, e.g., if (c,c) is a row in p.

Yet, brute-force evaluation is inefficient, inflexible and possibly unfeasible. For instance, if, in Example 9, (c,c) is a row in p, then the cause $p(c,c)$ of the violation of $\leftarrow p(x,x)$ in $D^{U'}$ is not tolerated if I' is checked brute-force. Of course, that is sound, since U' would increase the amount of violations.

However, in a database $D = ((T,V,IC),E)$ that is operational while undergoing schema updates, it may be impossible to delay operations until inserted constraints are evaluated entirely against possibly huge volumes of legacy data. Then, it should be advantageous to check an arbitrary schema update U by computing $\mathcal{M}(((T,V,IC^U),E),\, U_c)$. That is, inserted constraints are not evaluated brute-force, but are just checked against the update and future updates. Thus, updated constraints are not checked against legacy data, and the current and all future updates are prevented from introducing new causes of violation, at the expense of having to tolerate possible violations of inserted constraints by legacy data. If needed, they can be dealt with at less busy times, e.g., during night runs.

The following theorem reveals another advantage of using this policy: it may achieve completeness, while brute-force checking of inserted constraints cannot.

Theorem 3.
No method $\mathcal{M}$ that checks inserted constraints brute-force is complete for cause-based integrity checking.

Proof. If, in Example 9, no fact matching $p(a,y)$ is in D and $p(c,c)$ is the only fact that violates I', then, for each $\mathcal{M}$ that evaluates I' brute-force, $\mathcal{M}(D,U) = ko$ holds. However, condition (4) in Definition 6 warrants the output *ok*, since $U'_c = U$ clearly does not cause any violation of IC^U. Hence, $\mathcal{M}$ is not complete.

Example 10. (*Example 9, continued*).
If $\mathcal{M}$ checks U' by computing $\mathcal{M}(((T, V, IC^{U'}), E), U'_c)$, it will output *ok*, since $U'_c = U$ and I' is not relevant for U. Thus, causes such as $p(c, c)$ of the violation of I' in D are tolerated by $\mathcal{M}$.

Clearly, checking inserted constraints only with regard to U_c is much more efficient than evaluating them against the whole database. By the way, the latter is advocated in the SQL99 standard (cf. [19]), while the former lacks standardization, but is common practice.

4.4 View Modification and Unsatisfiability

Cause-based methods guarantee that all cases satisfied in the old state will remain satisfied in the new state, and that the set of causes of constraint violations does not grow larger. In fact, that continues to hold even if the given schema is unsatisfiable. Thus, inconsistency-tolerant methods are also unsatisfiability-tolerant. Hence, also the standard premise that the schema be satisfiable can be waived. We illustrate that by the following example where the modification of a view causes the unsatisfiability of the schema.

Example 11. Let $\{p(x, y) \leftarrow q(x, y), q(y, z);\ q(x, y) \leftarrow r(x, y);\ q(x, y) \leftarrow s(x, y)\}$ be the view definitions in D, and $IC = \{\leftarrow p(x, x);\ \exists\, x,y\ r(x, y)\}$. Clearly, the schema is satisfiable. Let (a, a), (a, b) be all tuples in s. Thus, $\leftarrow p(a, a)$ is a violated case. There are no more violated ground cases of $\leftarrow p(x, x)$ if and only if neither $r(b, a)$ nor any fact matching $r(x, x)$ is in D. Moreover, $\exists\, x,y\ r(x,y)$ is a violated case if and only if the extension of r is empty. Now, let U be the insertion of $q(x, y) \leftarrow r(y, x)$. Clearly, U makes the schema unsatisfiable. However, each inconsistency-tolerant method $\mathcal{M}$ can be soundly applied to check U for preserving satisfied cases. It is easy to see that $\mathcal{M}(D, U) = ko$ if and only if there is any tuple of form (A, B) in r such that (B, A) is not in r and (A, B) matches neither (b, a) nor (x, x). Otherwise, all cases of $\leftarrow p(x, x)$ that are violated in D^U are already violated in D, hence $\mathcal{M}(D, U) = ok$.

Although, in Example 11, the updated schema is unsatisfiable, it makes sense to accept further updates of D^U that do not violate any satisfied case. Such updates may even lower the number of violated cases, e.g., $U' = delete\ s(a, a)$.

4.5 Updating Safety-Critical Integrity Theories

As already seen, inconsistency tolerance is desirable. Yet, the need of safety-critical applications to have a set of ‘hard’ integrity constraints that are totally satisfied at all times should not be lightheartedly compromised. Hence, methods are called for that can guarantee total satisfaction of all hard constraints for dynamic schema maintenance. Fortunately, each inconsistency-tolerant method is capable of providing such a service. To see this, we first define the following reliability property, then infer Theorem 4 from it, and then interpret the definition and the result in terms of a dynamic maintenance of safety-critical applications.

Definition 7. $\mathcal{M}$ is called *reliable for hard constraints* if, for each pair (D, U) and each subset IC_h of IC such that each I in IC_h is a *hard* constraint, i.e., I must always be satisfied, i.e., $D(I) = sat$, the following implication holds.

$$\mathcal{M}(D,U) = ok \quad \Rightarrow \quad D^U(IC_h) = sat. \tag{5}$$

Theorem 4. Each cause-based method is reliable for hard constraints.

Proof. Let $\mathcal{M}$ be a cause-based method, and $IC_h \subset IC$. Then, (5) follows by applying Definition 6, property (3) and Definition 7.

By definition, each reliable method for hard constraints can maintain the total satisfaction of IC_h across updates, even if $IC \setminus IC_h$ is violated. However, for methods that are not inconsistency-tolerant, e.g., the one in [17], (5) may not hold, i.e., they may not be reliable. Example 12 shows that.

Example 12. Let $IC = \{\leftarrow q(a), r(x,x),\ \leftarrow q(b), r(x,x)\}$, $IC_h = \{\leftarrow q(b), r(x,x)\}$, and $D(IC_h) = sat$. Further, let $q(a)$ and $r(b,b)$ be the only facts in D that cause a violation of $\leftarrow q(a), r(x,x)$ in D, and $U = insert\ q(b)$. To check U, the method $\mathcal{M}_G$ in [17] drops $q(b)$ in $\leftarrow q(b), r(x,x)$, since U makes it *true*, thus obtaining the simplification $\leftarrow r(x,x)$. Since $\leftarrow q(a), r(x,x)$ is not relevant for U, the assumption of $\mathcal{M}_G$ that $D(IC) = sat$ wrongly entails that $D^U(\leftarrow q(a), r(x,x)) = sat$. Now, assume that D is distributed, q is locally accessible and r is remote. Then, $\mathcal{M}_G$ infers that also $D^U(\leftarrow r(x,x)) = sat$, since $q(a)$ is *true* in D^U. Hence, it unreliably outputs ok, although $D^U(IC_h) = vio$. As opposed to that, $\mathcal{M}(D,U) = ko$ holds for each cause-based inconsistency-tolerant method $\mathcal{M}$.

5 Related Work

Inconsistency tolerance is a subject of increasing importance [3]. Also inconsistency-tolerant integrity checking has received considerable attention recently [16]. The work in [16] is not based on causes, but on cases, i.e., instances of constraints that are relevant for given updates. In [8], it is shown that each case-based method is inconsistency-tolerant wrt causes, for updates of base facts and view definitions, but that the converse does not hold in general.

In general, causes are a smarter basis for inconsistency-tolerant integrity checking than cases, as argued in [7,8]. Among others, the concept of completeness is less problematic for causes than for cases. As outlined in [8,9], another advantage of causes over cases is that causes also provide a basis for computing answers that have integrity in inconsistent databases. The latter are similar to, though different from consistent query answers [2]. The relationship of the latter to cases is discussed in [9,16], and their relationship to causes in [8]. Moreover, causes offer advantages over cases with regard to partial repairs [12], and an automation of integrity checking for concurrent transactions [14].

Case-based inconsistency-tolerant schema updates are the theme of [6]. This paper upgrades the latter, by having causes take the place of cases, and by generalizing from definite to normal databases and constraints.

The concept of causes as proposed in [18] is much more complicated and involved than ours. The one in [18] is meant for explaining answers to human agents, but not, as in this paper, to programmed agents, e.g., modules that cater for integrity checking, repairing, query answering with integrity, or more.

The work in [5] is related to this paper by the topic of integrity updates. Like this paper, it is application-oriented, and additionally focuses also on issues such as database migration and web information systems, which we do not address, due to space limitation. However, inconsistency tolerance is not an issue in [5].

6 Conclusion

We have outlined how to extend conventional approaches to database schema update management. We have proposed a cause-based approach that can deal with arbitrary schema updates, including changes of the integrity theory. We have shown that such updates can be dealt with efficiently and reliably, without compromising hard integrity requirements for safety-critical applications.

As illustrated by Example 12, the advantages of our cause-based approach could not be obtained by employing any integrity checking method that is not inconsistency-tolerant. Thus, for updates U of integrity constraints, legacy data can safely be left alone only if the method used for checking U is inconsistency-tolerant.

Fortunately, the usual requirements of the total satisfaction of each database state and the satisfiability of the database schema can simply be waived, for most methods, without incurring any penalty. In fact, the power-to-mechanism ratio of our concept of inconsistency tolerance is exceedingly high, since no special mechanism is needed at all for achieving what has been disregarded by most researchers in the field until recently: inconsistency tolerance is provided for free by most methods, as argued in [15,16]. The main contribution of this paper is to have shown that cause-based inconsistency tolerance generalizes straightforwardly to updates than involve insertions or alterations of views and integrity constraints.

Ongoing work is concerned with applying a more general (not just case- or cause-based) metric approach to integrity checking of conventional updates, as presented in [13], also to schema updates. Moreover, we are working on the inconsistency-tolerant preservation of integrity across concurrent schema updates in distributed and replicated databases. A further topic of growing importance that is an objective of our current investigations is the metric-based control of consistency deterioration in cloud databases.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Arenas, M., Bertossi, L.E., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proceedings of PODS, pp. 68–79. ACM Press (1999)

3. Bertossi, L., Hunter, A., Schaub, T. (eds.): Inconsistency Tolerance. LNCS, vol. 3300. Springer, Heidelberg (2005)
4. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press (1978)
5. Curino, C., Moon, H., Deutsch, A., Zaniolo, C.: Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++. PVLDB 4, 117–128 (2010)
6. Decker, H.: Towards a dynamic inconsistency-tolerant schema maintenance. In: Song, I.-Y., et al. (eds.) ER Workshops 2008. LNCS, vol. 5232, pp. 89–98. Springer, Heidelberg (2008)
7. Decker, H.: Basic causes for the inconsistency tolerance of query answering and integrity checking. In: Proc. 21st DEXA Workshops, pp. 318–322. IEEE CSP (2010)
8. Decker, H.: Toward a uniform cause-based approach to inconsistency-tolerant database semantics. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2010. LNCS, vol. 6427, pp. 983–998. Springer, Heidelberg (2010)
9. Decker, H.: Answers that have integrity. In: Schewe, K.-D. (ed.) SDKB 2010. LNCS, vol. 6834, pp. 54–72. Springer, Heidelberg (2011)
10. Decker, H.: Causes for inconsistency-tolerant schema update management. In: Proc. 27th ICDE Workshops, pp. 157–161. IEEE CSP (2011)
11. Decker, H.: Consistent explanations of answers to queries in inconsistent knowledge bases. In: Roth-Berghofer, T., Tintarev, N., Leake, D. (eds.) Proc. IJCAI 2011 Workshop ExaCt, pp. 71–80 (2011)
12. Decker, H.: Partial repairs that tolerate inconsistency. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 389–400. Springer, Heidelberg (2011)
13. Decker, H.: Measure-based inconsistency-tolerant maintenance of database integrity. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2013. LNCS, vol. 7693, pp. 149–173. Springer, Heidelberg (2013)
14. Decker, H., de Marín, R.J.: Enabling business rules for concurrent transactions. In: Proc. Int. Conf. on P2P, Parallel, Grid, Cloud and Internet Computing, pp. 207–212. IEEE CPS (2001)
15. Decker, H., Martinenghi, D.: Classifying integrity checking methods with regard to inconsistency tolerance. In: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pp. 195–204. ACM Press (2008)
16. Decker, H., Martinenghi, D.: Inconsistency-tolerant integrity checking. IEEE Transactions of Knowledge and Data Engineering 23(2), 218–234 (2011)
17. Gupta, A., Sagiv, Y., Ullman, J.D., Widom, J.: Constraint checking with partial information. In: Proceedings of PODS 1994, pp. 45–55. ACM Press (1994)
18. Meliou, A., Gatterbauer, W., Moore, K., Suciu, D.: The complexity of causality and responsibility for query answers and non-answers. In: Proc. 37th VLDB, pp. 34–45 (2011)
19. Türker, C.: Schema evolution in SQL-99 and commercial (object-)relational DBMS. In: Balsters, H., De Brock, B., Conrad, S. (eds.) FoMLaDO 2000 and DEMM 2000. LNCS, vol. 2065, pp. 1–32. Springer, Heidelberg (2001)