INCONSISTENCY-TOLERANT ELIMINATIONS OF INTEGRITY VIOLATIONS

Hendrik Decker
Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Campus de Vera 8G, Valencia, Spain

Keywords: Inconsistency tolerance, Database integrity, Constraint violation, Repair.

Abstract: Violated integrity constraints should be repaired by eliminating the violations. However, it may be unfeasible to eliminate all violations. We show that it is possible to eliminate some but not necessarily all violations, i.e., to tolerate remaining inconsistencies, while preserving the consistent parts of the database.

1 INTRODUCTION

Database integrity can be enforced by repairing extant violations of integrity constraints. Traditionally, repairs are conceived to be *total*, i.e., all violations are supposed to be eliminated. In general, eliminating all violations is intractable (Chomicki, 2007). Thus, we propose to opt for *partial* repairs that are *inconsistency-tolerant* and *integrity-preserving*. For a database D and an integrity theory IC, such repairs eliminate some subset S of all constraint violations, while tolerating the persistence of violations in the complement of S, and preserving all consistent parts of the database.

In Section 2, we outline the formal background of the remainder; in particular, we recapitulate inconsistency-tolerant integrity checking (Decker and Martinenghi, 2011). In Section 3, we define repairs, and distinguish between partial and integrity-preserving repairs. Both partial and integrity-preserving repairs tolerate inconsistency, but only the latter guarantee the preservation of consistency. The theme of Section 4 is integrity-preserving repair management. Its goal is to show how to compute partial integrity-preserving repairs, in 4.3. For that purpose, we recapitulate integrity-preserving updating in 4.2. In Section 5, related work is addressed. In Section 6, we conclude with an outlook to further work.

2 BACKGROUND

After some preliminaries, we are going to recapitulate the concepts of inconsistency-tolerant integrity checking and integrity-preserving updates (Decker and Martinenghi, 2011). Unless specified otherwise, we use notations and terminology that are common for *datalog*.

As usual, we assume that each integrity constraint (in short, *constraint*) is represented as a *denial*, i.e., a clause of the form $\leftarrow B$, where the body B is a conjunction of literals that asserts what should not hold in any state of the database. An *integrity theory* is a finite set of constraints.

For an update U of a database D, we denote the updated database by D^U. For each sentence F, we write $D(F) = true$ (resp., $D(F) = false$) if F evaluates to *true* (resp., *false*) in D. Similarly, we write $D(I) = true$ (resp., $D(I) = false$) if I is satisfied (resp., violated) in D, and $D(IC) = true$ (resp., $D(IC) = false$) if all constraints in IC are satisfied in D (resp., at least one constraint in IC is violated in D).

Due to the possibly complex quantification of constraints, integrity checking tends to be prohibitively expensive, unless some simplification method is used (Christiansen and Martinenghi, 2006). Simplification theory traditionally requires that, for each update U, the state to be updated by U must satisfy all constraints. However, that requirement is unnecessary for inconsistency-tolerant integrity checking, as shown in (Decker and Martinenghi, 2011).

Integrity checking methods (in short, methods), and in particular inconsistency-tolerant ones, can be abstractly defined by their i/o behaviour. Each method $\mathcal{M}$ takes as input a database D, and integrity theory IC and an update U, and outputs either *ok* or *ko*. Intuitively, *ok* means that U does not increase the set of violated constraints, and *ko* that it may.

Decker H..
INCONSISTENCY-TOLERANT ELIMINATIONS OF INTEGRITY VIOLATIONS.
DOI: 10.5220/0003502503900395
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 390-395
ISBN: 978-989-8425-77-5

Definition 1. An *integrity checking method* maps triples (D, IC, U) to $\{ok, ko\}$. A method $\mathcal{M}$ is called *inconsistency-tolerant* if, for each database D, each integrity theory IC and each update U, $\mathcal{M}(D, IC, U) = ok$ entails that, for each constraint $I \in IC$ such that $D(I) = true$, also $D^U(I) = true$ holds.

Essentially, the only difference between conventional and inconsistency-tolerant integrity checking is that the former additionally requires total integrity before the update, i.e., that $D(IC) = true$. The absence of that requirement means that not necessarily each constraint has to be satisfied before D is updated, i.e., extant inconsistency in $D \cup IC$ is tolerated.

Note that Definition 1 is somewhat simpler than the original definition in (Decker and Martinenghi, 2011). The latter does not only require the preservation of the satisfaction of each constraint I such that $D(I) = true$, but of each instance I' of I such that $D(I') = true$. However, that difference is only a technical one, not an essential one, since each instance of each constraint can be taken as an individual element in IC. Indeed, it can be shown that Definition 1 is equivalent to the definition of the soundness of inconsistency-tolerant integrity checking in (Decker and Martinenghi, 2011).

3 REPAIRS

Repairing means to compute updates to databases in order to eliminate extant integrity violations. As already mentioned, repairing can be intractably costly. Thus, it should be a reasonable heuristic to curtail inconsistency by not repairing *all*, but only *some* violations, particularly in large databases with hidden or unknown inconsistencies.

The definition below, which is due to (Decker and Martinenghi, 2011), distinguishes between total repairs, which eliminate all inconsistencies, and partial repairs, which repair only a fragment of the database. Partial repairs tolerate inconsistency, since violated constraints in the complement of the repaired set may persist.

Definition 2. Let D be a database, IC an integrity theory and S a subset of IC such that $D(S) = false$. An update U is called a *repair* of S in D if $D^U(S) = true$. If $D^U(IC) = false$, U is also called a *partial repair* of IC in D. Otherwise, if $D^U(IC) = true$, U is called a *total repair* of IC in D.

In the literature, repairs usually are required to be total and minimal. Mostly, subset-minimality is opted for, but several other notions of minimality exist (Chomicki, 2007) or can be imagined. Note that Definition 2 does not involve any particular variant of minimality. However, Example 1 features subset-minimal repairs.

Example 1. Let $D = \{p(a,b,c),\ p(b,b,c),\ p(c,b,c),\ q(a,c),\ q(c,a),\ q(c,b),\ q(c,c)\}$ be a database and $IC = \{\leftarrow p(x,y,z) \wedge \sim q(x,z),\ \leftarrow q(x,x),\ \leftarrow p(x,y,y)\}$ an integrity theory. Clearly, the instances of constraints in IC that are violated in D are $\leftarrow p(b,b,c) \wedge \sim q(b,c)$ and $\leftarrow q(c,c)$. Hence, there are exactly two minimal total repairs of IC in D, viz. $\{delete\ q(c,c),\ delete\ p(b,b,c),\ delete\ p(c,b,c)\}$ and $\{delete\ q(c,c),\ insert\ q(b,c),\ delete\ p(c,b,c)\}$. Each of $U_1 = \{delete\ p(b,b,c)\}$ and $U_2 = \{insert\ q(b,c)\}$ is a minimal repair of $\{\leftarrow p(b,b,c) \wedge \sim q(b,c)\}$ in D and a partial repair of IC in D. Both tolerate the persistence of the violation of $\leftarrow q(c,c)$. Similarly, $U_3 = \{delete\ q(c,c)\}$ is a minimal repair of $\{\leftarrow q(c,c)\}$ in D and a partial repair of IC, which tolerates the violation of $\leftarrow p(b,b,c) \wedge \sim q(b,c)$.

A significant problem with partial repairs is that they may not preserve integrity, i.e., they may cause the violation of some constraint that is not in the repaired set, as shown by the following example.

Example 2. Consider again D and IC in Example 1. As opposed to U_1 and U_2, U_3 causes the violation of a constraint in the updated state that is satisfied before the update. That constraint is the instance $\leftarrow p(c,b,c) \wedge \sim q(c,c)$ of the first denial in IC. Thus, the non-minimal partial repair $U_4 = \{delete\ q(c,c);\ delete\ p(c,b,c)\}$ is needed to eliminate the violation of $\leftarrow q(c,c)$ in D without causing a violation that did not exist before the partial repair.

The enlargement of U_3 to U_4, i.e., deleting also $p(c,b,c)$, fortunately does not induce any similar side effect as produced by deleting $q(c,c)$ alone. In general, iterations such as the one from U_3 to U_4 may possibly continue indefinitely, due to iterative side effects. The termination of such iterations is unpredictable, in general, as is known from repairing by triggers (Ceri et al., 2000). However, such iterations can be avoided by checking if a given repair is an integrity-preserving update, according to the following definition.

Definition 3. For an integrity theory IC, an update U of a database D is called *integrity-preserving* if, for each instance I of any constrait in IC such that $D(I) = true$, also $D^U(I) = true$ holds.

Example 3. As seen in Example 2, both U_1 and U_2,

and also U_4, preserve integrity since all instances of constraints in *IC* that are satisfied in *D* remain satisfied in D^U. According to Definitions 2 and 3, the update U_4 is a minimal integrity-preserving repair of $\{\leftarrow q(x,x)\}$, although U_4 is not a mere minimal repair of $\{\leftarrow q(x,x)\}$, since the minimal repair U_3 of $\{\leftarrow q(x,x)\}$ is a proper subset of U_4. However, U_4 is preferable to U_3 since U_4 preserves integrity, while U_3 does not, as seen in Example 2. In general, each total repair (e.g., the two total repairs in Example 1) trivially preserves integrity, since no violations remain after total repairs.

4 INTEGRITY-PRESERVING REPAIR MANAGEMENT

In Section 3, we have distinguished desirable (partial) and preferable (integrity-preserving) repairs. However, all we have so far are definitions and examples, while a method to compute such repairs is still missing. The goal of this section is to close that void.

Fortunately, the main building blocks of the technology to compute partial and inconsistency-preserving repairs already exist. They recur on inconsistency-tolerant integrity checking (in short, *ITIC*), as characterized in Definition 1, and methods for computing integrity-preserving updates for satisfying given update requests, as discussed in (Decker and Martinenghi, 2011). In 4.1, we show how to check if repairs are integrity-preserving or not. In 4.2, we then recapitulate update computation. In 4.3, we finally show how update computation plus ITIC can compute partial and integrity-preserving repairs.

4.1 Checking Integrity Preservation

Clearly, each integrity-preserving update, hence each integrity-preserving repair, is inconsistency-tolerant, in the sense that there may be arbitrarily many constraint violations in *D* that persist in D^U. Moreover, the following result is an immediate consequence of Definitions 2 and 3.

Theorem 1. For each database *D*, each integrity theory *IC*, each update *U*, and each inconsistency-tolerant integrity checking method $\mathcal{M}$, *U* is integrity-preserving if $\mathcal{M}(D,IC,U) = ok$.

In general, the only-if version of Theorem 1 does not hold. However, it follows by definition that it does hold for methods that are *complete* for inconsistency-tolerant integrity checking, i.e., for methods that output *ok* whenever an update is integrity-preserving. For instance, the well-known method in (Nicolas, 1982) is complete for inconsistency-tolerant integrity checking, as shown in (Decker and Martinenghi, 2011).

Thus, Theorem 1 is important for the following reason: For each partial repair *U*, each inconsistency-tolerant integrity checking method can be used to check if *U* is integrity-preserving, and each complete inconsistency-tolerant method is a procedure for deciding if *U* is integrity-preserving or not.

4.2 Update Methods

We are going to define update methods as algorithms that take as input an update request and compute candidate updates as their output.

Definition 4.
a) An *update request* in a database *D* is a first-order sentence *R* that is to be made *true* by some integrity-preserving update *U*, i.e., $D^U(R) = true$ is requested to hold.
b) An update *U* is said to *satisfy* an update request *R* if $D^U(R) = true$ and *U* preserves integrity. Clearly, view update requests are a well-known special kind of update requests.
c) An *update method* is an algorithm that, for each database *D* and each update request *R*, computes candidate updates $U_1, \ldots, U_n$ $(n \geq 0)$ such that $D^{U_i}(R) = true$ $(1 \leq i \leq n)$.

A well-known special case of update requests are view update requests. Essentially, a view update request is expressed by a literal whose predicate is not a base relation but a database view predicate. It is to be satisfied, i.e., to be made *true*, by an update of the base relations by which the view predicate is defined. Thus, the class of methods for computing view update requests is a special case of update methods.

Note that, according to Definition 4*c*, an update method is impartial with regard to any integrity violation that may be caused by any of the U_i. As opposed to that, Definition 5, below, is going to take such undesirable side effects into account.

To avoid that updates cause new integrity violations, many of the known update methods in the literature (e.g., (Decker, 1990; Guessoum and Lloyd, 1990; Kakas and Mancarella, 1990)) postulate the total satisfaction of all constraints in the state before the update, in analogy to the total integrity premise of traditional integrity checking, as mentioned after Definition 1. However, that requirement is as superfluous for satisfying update requests as for integrity checking, for the class of update methods defined next.

Definition 5. An update method $\mathcal{UM}$ is *integrity-preserving* if each update computed by $\mathcal{UM}$ preserves integrity.

For an update request R and a database D, several update methods in the literature work by two separate phases. First, a candidate update U such that $D^U(R) = true$ is computed. Then, U is checked for integrity preservation by some inconsistency-tolerant integrity checking method. If that check is positive, U is accepted. Else, U is rejected and another candidate update, if any, is computed and checked. Hence, Theorem 2, below, follows from the definitions above.

Theorem 2. Each update method that uses an inconsistency-tolerant method to check its computed candidate updates is integrity-preserving.

Theorem 2 serves to identify several known update methods as integrity-preserving, since they use inconsistency-tolerant integrity checking. Among them are the update methods described in (Decker, 1990) and (Guessoum and Lloyd, 1990). Several other known update methods are abductive e.g., (Kakas and Mancarella, 1990; Kakas et al., 1998; Dung et al., 2006). They somehow interleave the two phases as addressed above. Most of them are also integrity-preserving, as has been shown in (Decker and Martinenghi, 2011) for the method in (Kakas and Mancarella, 1990).

The following example illustrates the usefulness of integrity-preserving update methods, by featuring what can go wrong if an update method that is not integrity-preserving is used.

Example 4. 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 $U = \{insert\ r(a),\ insert\ s(a)\}$. To check if U preserves integrity, most 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 (e.g., those in (Gupta et al., 1994; Lee and Ling, 1996)) may be mislead to use the invalid premise that $D(IC) = true$, by reasoning as follows.

The 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 wrongly conclude that U preserves integrity, since the case $\leftarrow p(a,y) \wedge q(y)$ is satisfied in D but violated in D^U. By contrast, each inconsistency-tolerant method rejects U and computes the update $U' = U \cup \{delete\ p(a,a)\}$ for satisfying R. Clearly, U' preserves integrity. Note that, incidentally, U' even removes the violated case $\leftarrow p(a,a)$.

In fact, the reduction of the amount of inconsistency as observed in Example 4 is not accidental. In general, as long as inconsistency-tolerant integrity checking is applied for each update, the number of violated cases is not only prevented from increasing, but also is likely to decrease over time, since each update, be it accidentally or on purpose, may repair part or all of the extant inconsistencies.

4.3 Computing Partial Repairs that are Integrity-preserving

The following example illustrates a general approach of how partial repairs can be computed by update methods off the shelve.

Example 5. Let $S = \{\leftarrow B_1, \ldots, \leftarrow B_n\}$ $(n \geq 0)$ be a set of cases of constraints in an integrity theory IC of a database D. Thus, $D(S) = false$ if and only if $D(B_i) = true$ for some i. Further, suppose that there is a case in $IC \setminus S$ that is violated in D. Hence, a partial repair can be computed by each update method, simply by issuing the update request $\sim vio_S$, where vio_S be defined by the clauses $vio_S \leftarrow B_i$ $(1 \leq i \leq n)$.

Now we recall from Section 3 that partial repairs may not preserve integrity. That problem is solved by the following consequence of Theorems 1 and 2. It says that the integrity preservation of partial repairs can be checked by inconsistency-tolerant integrity checking (part *a*), and that integrity-preserving repairs can be computed by integrity-preserving update methods (part *b*).

Theorem 3.

a) For each database D, each integrity theory IC, each partial repair U of IC in D and each inconsistency-tolerant method $\mathcal{M}$ such that $\mathcal{M}(D, IC, U) = ok$, U is integrity-preserving.

b) Each partial repair computed as in Example 5 with an integrity-preserving update method is integrity-preserving.

5 RELATED WORK

Traditionally, concepts of repair in the literature (e.g., in (Arenas et al., 1999; Greco et al., 2003; Eiter et al., 2008)) only deal with total repairs. To the best of the author's knowledge, partial repairs have never been addressed elsewhere, except in (Decker and Martinenghi, 2011). In (Furfaro et al., 2007), null values

and a 3-valued semantics are used to "summarize" total repairs. Since integrity preservation is a trivial issue for total repairs, there is also no notion of integrity-preserving updates or repairs in the literature.

Total repairs can be exceedingly costly, and so can partial repairs, in general. However, by comparison, partial repairs are more feasible than total repairs, simply because the violations of some integrity constraints may be hidden, unknown or not resolvable, while the repair of the violation of others may be fairly straightforward. Moreover, the application of our definitions and results is not compromised by any limitation with regard to the syntax of integrity constraints, while severe syntactical restrictions are typical in the literature on repairs.

A broadly discussed issue in the literature about repairs is repair checking, i.e., algorithms for deciding if a given update is a repair or not. Analogous to similar definitions in (Chomicki, 2007; Afrati and Kolaitis, 2009), the problem of *integrity-preserving partial repair checking* can be defined as the check if a given update is an integrity-preserving repair. Thus, Theorem 3*a* entails that each inconsistency-tolerant integrity checking method is an implementation of inconsistency-tolerant repair checking.

Probably the most widely discussed topic related to repairs is consistent query answering (CQA) (Arenas et al., 1999). It defines an answer to be consistent in a database D with regard to an integriy theory IC if it is *true* in each minimal repair of IC in D. CQA suffers from its dependence on the chosen notion of minimality, of which our definitions are steered clear. Moreover, CQA usually is not computed by computing each repair, but by techniques of semantic query optimization or disjunctive logic programming. It should be interesting to devise a new way of computing CQA by computing partial instead of total repairs, since, in general, not all violated constraints are relevant with regard to the given query.

6 CONCLUSIONS

The evolution of a database typically involves fallacious updates and other events that may compromise the quality of the stored data, e.g., during down- and uploads, migrations, changes in the schema, system failures, etc. In particular, it is hard to avoid that some violations of integrity constraints occur and persist. Thus, the need for a systematic maintenance of the quality of the stored data arises. One way to meet that challenge is to eliminate extant violations of integrity constraints. Since a total elimination of all inconsistencies is intractable, in general, the need to tolerate inconsistency imposes itself as well.

In this paper, we have presented an approach to reconcile the conflict between eliminating and living with integrity violations in databases. It consists in opting for possibly partial repairs, instead of total repairs. Partial repairs are inconsistency-tolerant, in the sense that only some but not all causes of integrity violations are eliminated, while violations of constraints not included in the repaired subset may remain inconsistent. As illustrated by a paradigmatic example, partial repairs can be computed by any method for view updating.

A severe problem with partial repairs is that they may have the unpleasant side effect of increasing the amount of inconsistency in the fragment of the database that is not repaired. In order to avoid that problem, inconsistency-tolerant updates that are integrity-preserving need to be filtered out of the set of candidate partial repairs. To do that, the updates associated to partial repairs should be checked for integrity preservation.

Traditionally, integrity checking methods had been believed to be not applicable for checking updates for inconsistency-tolerant integrity preservation. They all have insisted on the requirement of total consistency, which cannot be complied with whenever repairs are partial. Fortunately, however, many known integrity checking methods could be shown to be inconsistency-tolerant (Decker and Martinenghi, 2011), and hence applicable to check partial repairs for integrity preservation.

Future work is concerned with replacing the notion of cases by a similar but more basic notion of causes, for explaining the reasons for integrity violations. Causes provide a uniform basis for an alternative concept of inconsistency tolerance and, at the same time, of 'answers that have integrity' (AHI) (Decker, 2010). The latter is not provided by case-based ITIC. Based on causes, AHI is, by intents and purposes, similar to CQA, and, as argued in (Decker, 2010), compares favorably to CQA.

Replacing repairs of violated cases by repairs of the actual causes of integrity violation is going to be elaborated in a follow-up version of this paper.

ACKNOWLEDGEMENTS

This work has been partially supported by FEDER and the Spanish grants TIN2009-14460-C03, TIN2010-17139.

REFERENCES

Afrati, F. and Kolaitis, P. (2009). Repair checking in inconsistent databases: algorithms and complexity. In *Proc. 12th ICDT*, pages 31–41. ACM Press.

Arenas, M., Bertossi, L. E., and Chomicki, J. (1999). Consistent query answers in inconsistent databases. In *Proc. 18th PODS*, pages 68–79. ACM Press.

Ceri, S., Cochrane, R., and Widom, J. (2000). Practical applications of triggers and constraints: Success and lingering issues (10-year award). In In *Proc. 26th VLDB*, pages 254–262. Morgan Kaufmann.

Chomicki, J. (2007). Consistent query answering: Five easy pieces. In *Proc. 11th ICDT*, volume 4353 of *LNCS*, pages 1–17. Springer.

Christiansen, H. and Martinenghi, D. (2006). On simplification of database integrity constraints. *Fundamenta Informaticae*, 71(4):371–417.

Decker, H. (1990). Drawing updates from derivations. In *Proc. 3rd ICDT*, volume 470 of *LNCS*, pages 437–451. Springer.

Decker, H. (2010). Toward a uniform cause-based approach to inconsistency-tolerant database semantics. In *Proc. 9th ODBASE, Part II*, volume 6427 of *LNCS*, pages 983–998. Springer.

Decker, H. and Martinenghi, D. (2011). Inconsistency-tolerant integrity checking. *IEEE Transactions of Knowledge and Data Engineering*, 23(2):218–234.

Dung, P. M., Kowalski, R., and Toni, F. (2006). Dialectic proof procedures for assumption-based admissible argumentation. *Artif. Intell.*, 170(2):114–159.

Eiter, T., Fink, M., Greco, G. and Lembo, D. (2008). Repair localization for query answering from inconsistent databases. *ACM Transactions of Database Systems*, 33(2).

Furfaro, F., Greco, S. and Molinaro, C. (2007). A three-valued semantics for querying and repairing inconsistent databases. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):167–193.

Greco, G., Greco, S. and Zumpano, E. (2003). A logical framework for querying and repairing inconsistent databases. *IEEE Transactions of Knowledge and Data Engineering*, 15(6):1389–1408.

Guessoum, A. and Lloyd, J. (1990). Updating knowledge bases. *New Generation Computing*, 8(1):71–89.

Gupta, A., Sagiv, Y., Ullman, J. D., and Widom, J. (1994). Constraint checking with partial information. In *Proc. 13th PODS*, pages 45–55. ACM Press.

Kakas, A. and Mancarella, P. (1990). Database updates through abduction. In *Proc. 16th VLDB*, pages 650–661. Morgan Kaufmann.

Kakas, A., Kowalski, R., and Toni, F. (1998). The role of abduction in logic programming. *Handbook in Artificial Intelligence and Logic Programming*, 5:235–324.

Lee, S. Y. and Ling, T. W. (1996). Further improvements on integrity constraint checking for stratifiable deductive databases. In *Proc. 22nd VLDB*, pages 495–505. Kaufmann.

Nicolas, J.-M. (1982). Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253.