Cuzzocrea, A.; Decker, H.; Muñoz-Escoí, FD. (2013). Capturing and Scaling Up Concurrent Transactions in Uncertain Databases. Communications in Computer and Information Science. 246:70-85. doi:10.1007/978-3-642-42017-7_6.

Capturing and Scaling-Up Concurrent Transactions over Uncertain Databases

Alfredo Cuzzocrea[1], Hendrik Decker[2***], and Francesc D. Muñoz-Escoí[2†]

Abstract This chapter provides *a complete framework for capturing and scaling-up concurrent transactions over uncertain databases*. Models and methods proposed in the context of this framework for *managing data uncertainty* are innovative as previous studies have not considered the specific case of *concurrent transactions*, which may worse the uncertainty of database management activities beyond to the simplest case of *isolated transaction*s. Indeed, as this chapter demonstrates, *inconsistency tolerance of integrity management*, *constraint checking* and *repairing* easily scale up to concurrent transactions in a natural way, and query answers in concurrent transactions over uncertain data remain *certain* in the presence of uncertainty. This analytical contribution is enriched by means of a *reference architecture* for uncertain database management under concurrent transactions that strictly adheres to models and methods that are the main contributions of this research.

Key words: Uncertain Databases, Concurrent Transactions, Inconsistence Tolerance of Integrity Management

1 Introduction

Uncertainty in databases is closely related to *inconsistency*, i.e. *lack of integrity*, in two ways. Firstly, the validity of answers in inconsistent databases obviously is uncertain. Secondly, conditions for stating properties of uncer-

*** Supported by FEDER and the Spanish grants TIN2009-14460-C03, TIN2010-17139.

† Supported by FEDER and the Spanish grants TIN2009-14460-C03, TIN2010-17139.

ICAR-CNR and University of Calabria, I-87036 Cosenza, Italy · Instituto Tecnológico de Informática, UPV, E-46022 Valencia, Spain

tainty of data can be modeled as integrity constraints. Thus, each constraint violation corresponds to some uncertainty in the database, no matter if the constraint models a regular integrity assertion or some specific uncertainty condition. This paper addresses both of the mentioned relations between uncertainty and inconsistency.

For instance, the denial $\leftarrow item(x, y),\ y < 75\%$ constrains entries x in the *item* table to have a probability (certainty) y of at least 75%. Similarly, the constraint $I = \leftarrow uncertain(x)$, where *uncertain* is defined by the database clause $uncertain(x) \leftarrow email(x, from(y)),\ {\sim}authenticated(y)$, bans each email message x that is uncertain because its sender y has not been authenticated. Likewise, *uncertain* could be defined, e.g., by $uncertain(x) \leftarrow item(x, null)$, indicating an uncertainty about each item x the attribute of which has a null value.

An advantage of representing uncertainty by constraints is that the evolution of uncertainty across updates can then be monitored by inconsistency-tolerant methods for integrity checking, and uncertainty can then be eliminated by integrity repairing. For instance, each update U that tries to insert an email by a non-authenticated sender will be rejected by each method that checks U for integrity preservation, since U would violate I, in the preceding example. Ditto, stored email entries with unauthenticated senders or items with unknown attributes can be eliminated by repairing the violations of I in the database.

Conventional approaches to integrity management unrealistically require total constraint satisfaction before an update is checked and after a repair is done. However, methods for checking or repairing integrity or uncertainty must be inconsistency-tolerant as soon as data that violate some constraint are admitted to persist across updates. In [22], we have shown that the total consistency requirement can be waived without further ado for most (though not all) known methods. Thus, they can be soundly applied in databases with persistent constraint violations, i.e. with extant inconsistency and uncertainty.

Rather than pretending that consistent databases certainly remain consistent across updates (as conventional methods do), inconsistency-tolerant methods just assure that inconsistency, i.e., uncertainty is not increased, neither by updates nor by repairs. Such increase or decrease is determined by violation measures [20] (called 'inconsistency metrics' in [19]). Some of these measures also serve to provide answers that have integrity in the presence of uncertainty, by adopting an inconsistency-tolerant approach proposed in [18], called AHI.

Inconsistency tolerance also enables uncertainty management for concurrent transactions. For making any guarantees of integrity preservation across concurrent transactions, the usual requirement is that each transaction maps each consistent state to a consistent successor state. Unfortunately, that excludes any prediction for what is going to happen in the presence of constraint violations, i.e., of uncertainty. However, we are going to see that the incon-

sistency tolerance of integrity management easily scales up to concurrent transactions, and concurrent query answering with AHI remains certain in the presence of uncertainty.

After some preliminaries in Section 2, we recapitulate inconsistency-tolerant integrity management (checking, repairing and query answering) in Section 3. In Section 4, we elaborate an example of how to manage uncertainty expressed by constraints. In Section 5, we outline how inconsistency-tolerant constraint management scales up to database systems with concurrent transactions. In Section 6, we provide a reference architecture for uncertain database management under concurrent transactions, which incorporates our models and algorithms. In Section 7, we address related work. In Section 8, we conclude and provide future research directions.

2 Formal Terminology and Definitions

We use terminology and formalisms that are common for *datalog* [1]. Also, we assume some familiarity with transaction concurrency control [6].

Throughout the paper, we use symbols like D, I, IC, U for representing a database, an integrity constraint (in short, *constraint*), a finite set of constraints (also called *integrity theory*) and, resp., an update. We denote the result of executing an update U on D by D^U, and the truth value of a sentence or a set of sentences S in D be denoted by $D(S)$.

Constraints often are asserted as denials, i.e., clauses with empty head of the form $\leftarrow B$, where the body B is a conjunction of literals that state what should not be true in any state of the database. For each constraint I that expresses what should be true, a denial form of I can be obtained by re-writing $\leftarrow \sim I$ in clausal form, as described, e.g., in [17]. Instead of leaving the head of denial constraints empty, a predicate that expresses some lack of consistency may be used in the head. For instance, $uncertain \leftarrow B$ explicitly states an uncertainty that is associated to each instance of B that is true in the database.

3 Management of Uncertainty that Tolerates Inconsistency

As argued in Section 1, violations of constraints, i.e., the inconsistency of given database states with their associated integrity theory, reflect uncertainty. Each update may violate or repair constraints, and thus increase or decrease the amount of uncertainty. Hence, checking updates for such increases, and decreasing uncertainty by reparing violated constraints, are essential for uncertainty management. Also mechanisms for providing answers

that are certain in uncertain databases are needed. In 3.1, 3.2, and 3.3, we recapitulate and extend measure-based inconsistency-tolerance for integrity checking [19], repairing [20] and, resp., query answering [18], in terms of uncertainty.

3.1 Measure-based Uncertainty-tolerant Integrity Checking

The integrity constraints of a database are meant to be checked upon each update, which usually is committed only if it does not violate any constraint. Since total integrity is rarely achieved, and in particular not in databases where uncertainty is modeled by constraints, integrity checking methods that are able to tolerate uncertainty are needed.

In [22], inconsistency-tolerant integrity checking has been formalized and discussed. In particular, it has been shown that many (but not all) existing integrity checking methods tolerate inconsistency and thus uncertainty, although most of them have been designed to be applied only if all constraints are totally satisfied before any update is checked. In [19], we have seen that integrity checking can be described by 'violation measures' [20], which are a form of inconsistency measures [27]. Such measures, called 'uncertainty measures' below, size the amount of violated constraints in pairs (D, IC). Thus, an update can be accepted if it does not increase the measured amount of constraint violations.

Definition 1. *We say that* $(\mu, \preccurlyeq)$ *is an* uncertainty measure *(in short, a* measure*) if* μ *maps pairs* (D, IC) *to some metric space* $(\mathbb{M}, \preccurlyeq)$ *where* $\preccurlyeq$ *is a partial order, i.e. a binary relation on* $\mathbb{M}$ *that is antisymmetric, reflexive and transitive. For* $E, E' \in \mathbb{M}$*, let* $E \prec E'$ *denote that* $E \preccurlyeq E'$ *and* $E \neq E'$*.*

In [25, 27] and [19, 20], various axiomatic properties of uncertainty measures that go beyond Definition 1 are proposed. Here, we refrain from that, since the large variety of conceivable inconsistency measures has been found to be "too elusive to be captured by a single definition" [25]. Moreover, several properties that are standard in measurement theory [4] and that are postulated also for inconsistency measures in [25, 27] do not hold for uncertainty measures, due to the non-monotonicity of database negation, as shown in [20].

Definition 2 captures each integrity checking method $\mathcal{M}$ (in short, method) as an I/O function that maps updates to $\{ok, ko\}$. The output *ok* means that the checked update is acceptable, and *ko* that it may not be acceptable. For deciding to *ok* or *ko* an update, $\mathcal{M}$ uses an uncertainty measure.

Definition 2. (Uncertainty-tolerant Integrity Checking (abbr.: UTIC))
An integrity checking method *maps triples* (D, IC, U) *to* $\{ok, ko\}$*. Each such*

method $\mathcal{M}$ is called a sound *(resp.,* complete*)* UTIC method *if there is an uncertainty measure $(\mu, \preccurlyeq)$ such that, for each (D, IC, U), (1) (resp., (2)) holds.*

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

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

If $\mathcal{M}$ is sound, it is also called a μ-based UTIC method.

The only real difference between conventional integrity checking and UTIC is that the former additionally requires total integrity before the update, i.e., that $D(IC) = true$ in the premise of Definition 2. The range of the measure μ used by conventional methods is the binary metric space $(\{true, false\}, \preccurlyeq)$ where $\mu(D, IC) = true$ means that IC is satisfied in D, $\mu(D, IC) = false$ that it is violated, and $true \prec false$, since, in each consistent pair (D, IC), there is a zero amount of uncertainty, which is of course less than the amount of uncertainty of each inconsistent pair (D, IC).

More differentiated uncertainty measures are given, e.g., by comparing or counting the sets of instances of violated constraints, or the sets of 'causes' of inconsistencies. Causes (characterized more precisely in 3.3.1) are defined in [18, 19] as the data whose presence or absence in the database is responsible for integrity violations. Other violation measures are addressed in [19].

As seen in [19], many conventional methods can be turned into measure-based uncertainty-tolerant ones, simply by waiving the premise $D(IC) = true$ and comparing violations in (D, IC) and (D^U, IC). If there are more violations in (D^U, IC) than in (D, IC), they output *ko*; otherwise, they may output *ok*. According to [22], the acceptance of U by an uncertainty-tolerant method guarantees that U does not increase the set of violated instances of constraints.

More generally, the following result states that uncertainty can be monitored and its increase across updates can be prevented by each UTIC method, in as far as uncertainty is modeled in the syntax of integrity constraints.

Theorem 1. Let D be a database and IC an integrity theory that models uncertainty in D. Then, the increase of uncertainty in D by any update U can be prevented by checking U with any sound UTIC method.

3.2 Uncertainty-tolerant Integrity-preserving Repairs

In essence, repairs consist of updates that eliminate constraint violations [30]. However, hidden or unknown violations may be missed when trying to repair a database. Moreover, as known from repairing by triggers [10], updates that eliminate some violation may inadvertedly violate some other constraint. Hence, uncertainty-tolerant repairs are called for. Below, we recapitulate the definition of partial and total repairs in [22]. They are uncertainty-tolerant

since some violations may persist after partial repairs. But they may not preserve integrity.

Definition 3. (Repair)
For a triple (D, IC, U)*, let* S *be 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*.*

Example 1. Let $D = \{p(1,2,3),\ p(2,2,3),\ p(3,2,3),\ q(1,3), q(3,2),\ q(3,3)\}$ and $IC = \{\leftarrow p(x,y,z) \wedge \sim q(x,z),\ \leftarrow q(x,x)\}$. Clearly, both constraints are violated. $U = \{delete\, q(3,3)\}$ is a repair of $\{\leftarrow q(3,3)\}$ in D and a partial repair of IC. It tolerates the uncertainty reflected by the violation of $\leftarrow p(2,2,3) \wedge \sim q(2,3)$ in D^U. However, U also causes the violation of $\leftarrow p(3,2,3) \wedge \sim q(3,3)$ in D^U. Thus, the partial repair $U' = \{delete\, q(3,3),\ delete\, p(3,2,3)\}$ is needed to eliminate the violation of $\leftarrow q(3,3)$ in D without causing any other violation.

Example 1 illustrates the need to check if a given update or partial repair is *integrity-preserving*, i.e., does not increase the amount of uncertainty. This problem is a generalization of what is known as *repair checking* [2]. The problem can be solved by UTIC, as stated in Theorem 2.

Theorem 2. Let $(\mu, \preccurlyeq)$ be an uncertainty measure, $\mathcal{M}$ a UTIC method based on $(\mu, \preccurlyeq)$, and U a partial repair of IC in D. For a tuple (D, IC), U preserves integrity wrt. μ, i.e., $\mu(D^U, IC) \preccurlyeq \mu(D, IC)$, if $\mathcal{M}(D, IC, U) = ok$.

For computing partial repairs, any off-the-shelve view update method can be used, as follows. Let $S = \{\leftarrow B_1, \ldots, \leftarrow B_n\}$ be a subset of constraints to be repaired in a database D. Candidate updates for satisfying the view update request can be obtained by running the view update request *delete violated* in $D \cup \{violated \leftarrow B_i \mid 0 \leq i \leq n\}$. For deciding if a candidate update U preserves integrity, U can be checked by UTIC, according to Theorem 2.

3.3 Certain Answers in Uncertain Databases

Violations of constraints that model uncertainty may impair the integrity of query answering, since the same data that cause the violations may also cause the computed answers. Hence, there is a need of an approach to provide answers that either have integrity and thus are certain, or that tolerate some uncertainty. An approach to provide answers that are certain in uncertain databases is outlined in 3.3.1, and generalized in 3.3.2 to provide answers that tolerate uncertainty.

3.3.1 Answers that are certain

Consistent query answering (abbr. *CQA*) [3] provides answers that are correct in each minimal total repair of IC in D. CQA uses semantic query optimization [11] which in turn uses integrity constraints for query answering. A similar approach is to abduce consistent hypothetical answers, together with a set of hypothetical updates that can be interpreted as integrity-preserving repairs [24].

A new approach to provide answers that have integrity (abbr. AHI) and thus certainty is proposed in [18]. AHI determines two sets of data: the causes by which an answer is deduced, and the causes that lead to constraint violations. More precisely, for databases D and queries without negation in the body of clauses, causes are minimal subsets of ground instances of clauses in D by which positive answers or violations are deduced. For clauses with negation in the body and negative answers, also minimal subsets of ground instances of the only-if halves of the if-and-only-if completions of predicates in D [12] form part of causes.

An answer then is defined to have integrity if it has a cause that does not intersect with any of the causes of constraint violations, i.e., if it is deducible from data that are independent of those that violate constraints. Definition 4 below is a compact version of the definition of AHI in terms of certainty. Precise definitions of causes and details of computing AHI are in [18, 19, 20].

Definition 4. *Let θ be an answer to a query $\leftarrow B$ in (D, IC), i.e., θ is either a substitution such that $D(\forall(B\theta)) = true$ or $D(\leftarrow B) = true$, i.e., $\theta = \text{no}$.*

a) *Let B_θ stand for $\forall(B\theta)$ if θ is a substitution, or for $\leftarrow B$ if $\theta = \text{no}$.*

b) θ is certain *in (D, IC) if there is a cause C of B_θ in D such that $C \cap C_{IC} = \emptyset$, where C_{IC} is the union of all causes of constraint violations in (D, IC).*

3.3.2 Answers that tolerate uncertainty

AHI is closely related to UTIC, since some convenient violation measures are defined by causes: cause-based methods accept an update U only if U does not increase the number or the set of causes of constraint violations [19]. Similar to UTIC, AHI is uncertainty-tolerant since it provides correct results in the presence of constraint violations. However, each answer accepted by AHI is independent of any inconsistent parts of the database, while UTIC may admit updates that violate constraints. For instance, U in Example 1 causes the violation of a constraint while eliminating some other violation. Now, suppose U is checked by some UTIC method based on a violation measure that assigns a greater weight to the eliminated violation than to the newly

caused one. Thus, U can be *ok*-ed, since it decreases the measured amount of inconsistency.

In this sense, we are going to relax AHI to ATU: answers that tolerate uncertainty. ATU sanctions answers that are acceptable despite some amount of uncertainty involved in their derivation.

To quantify that amount, some 'tolerance measure' is needed. Unlike uncertainty measures which size the amount of uncertainty in all of (D, IC), tolerance measures only size the amount of uncertainty involved in the derivation of given answers or violations.

Definition 5. (ATU)
a) *For answers θ to queries $\leftarrow B$ in (D, IC), a* tolerance measure *maps triples (D, IC, B_θ) to $(\mathbb{M}, \preceq)$, where $\mathbb{M}$ is a metric space partially ordered by $\preceq$.*
b) *Let* th *be a threshold value in $\mathbb{M}$ up to which uncertainty is tolerable. Then, an answer θ to some query $\leftarrow B$ in (D,IC) is said to* tolerate uncertainty up to th *if $\tau(D, IC, B_\theta) \preceq$ th .*

A first, coarse tolerance measure τ could be to count the elements of $C_\theta \cap C_{IC}$ where C_θ is the union of all causes of B_θ and C_{IC} is as in Definition 4. Or, taking application semantics into account, a specific weight may be assigned to each element of each cause, similar to the tuple ranking in [5]. Then, τ can be defined by adding up the weights of elements in $C_\theta \cap C_{IC}$. Another possibility to define τ: application-specific weights could be assigned to each ground instance I' of each $I \in IC$. Then, τ could sum up the weights of those I' that have a cause C' such that $C_\theta \cap C' \neq \emptyset$.

For example, $\tau(D, IC, B_\theta) = |\, C_{theta} \cap C_{IC} \,|$ counts elements in $C_\theta \cap C_{IC}$, where $|\,.\,|$ is the cardinality operator. Or, $\tau(D,IC,B_\theta) = \sum \{\, \omega(c) \mid c \in C_\theta \cap C_{IC}\}$ adds up the weights of elements in $C_\theta \cap C_{IC}$, where ω is a weight function.

4 Uncertainty Management in Concurrent Database Transactions – An Example

In this section, we illustrate the management of uncertainty by inconsistency-tolerant integrity management, and discuss some more conventional alternatives. In particular, we compare uncertainty-tolerant integrity management with brute-force constraint evaluation, conventional integrity checking that is not uncertainty-tolerant, total repairing, and CQA, in 4.1 – 4.6.

The predicates and their attributes below are open to interpretation. By assigning convenient meanings to predicates, it can be interpreted as a model of uncertainty in a decision support systems for, e.g., stock trading, or controlling operational hazards in a complex machine.

Let D be a database with the following definitions of view predicates *ul*, *um*, *uh* that model uncertainty of low, medium and, respectively, high degree:

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

$um(y) \leftarrow q(x,y),\ {\sim}p(y,x)$; $\quad um(y) \leftarrow p(x,y),\ q(y,z),\ {\sim}p(y,z),\ {\sim}q(z,x)$

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

where th be a threshold value greater or equal 0. Now, let uncertainty be denied by the following integrity theory:

$IC\ =\ \{\leftarrow ul(x),\ \ \leftarrow um(x),\ \ \leftarrow uh(x)\}$.

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

$p(0,0),\ p(0,1),\ p(0,2),\ p(0,3), \ldots, p(0,10000000),$

$p(1,2), p(2,4), p(3,6), p(4,8), \ldots, p(5000000, 10000000)$

$q(0,0),\ q(1,0),\ q(3,0),\ q(5,0),\ q(7,0), \ldots, q(9999999,0)$

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

Now, let us consider the update $U = \mathit{insert}\ q(0, 9999999)$.

4.1 Brute-force Uncertainty Management

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

For large extensions of p and q, brute-force evaluation of IC clearly may last too long, in particular for safety-critical uncertainty monitoring in real time. In 4.2, we are going to see that it is far less costly to use an UTIC method that simplifies the evaluation of constraints by confining its focus on the data that are relevant for the update.

4.2 Uncertainty Management by UTIC

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

At update time, the following simplifications of medium and high uncertainty constraints are obtained from U. (No low uncertainty is caused by

U since $q(0, 9999999)$ does not match $p(x, x)$.) These simplifications are obtained at hardly any cost, by simple pattern matching of U with pre-simplified constraints that can be compiled at constraint specification time.

$\leftarrow \sim p(9999999, 0)$; $\quad \leftarrow p(x, 0),\ \sim p(0, 9999999),\ \sim q(9999999, x)$

$\leftarrow p(0, 0),\ 9999999 > th$

By a simple lookup of $p(9999999, 0)$ for evaluating the first of the three denials, it is inferred that $\leftarrow um$ is violated.

Now that a medium uncertainty has been spotted, there is no need to check the other two simplifications. Yet, let us do that, for later comparison in 4.3.

Evaluation of the second simplification from left to right essentially equals the cost of computing the answer $x = 0$ to the query $\leftarrow p(x, 0)$ and successfully looking up $q(9999999, 0)$. Hence, the second denial is *true*, i.e., there is no further medium uncertainty. Clearly, the third simplification is violated if $9999999 > th$ holds, since $p(0, 0)$ is true, i.e., U possibly causes high uncertainty.

Now, let us summarize this subsection. Validating U by UTIC according to Theorem 1 essentially costs a simple access to the p relation. Only one more look- up is needed for evaluating all constraints. And, apart from a significant cost reduction, UTIC prevents medium and high uncertainty constraint violations that would be caused by U if it were not rejected.

4.3 Methods that are Uncertainty-intolerant

UTIC is sound, but, in general, methods that are uncertainty-intolerant (i.e., not uncertainty-tolerant, e.g., those in [26, 28]) are unsound, as shown below.

Clearly, p is not affected by U. Thus, $D(ul(x)) = D^U(ul(x))$. Each integrity checking method that is uncertainty-intolerant assumes $D(IC) = true$. Thus, the method in [26] concludes that the unfolding $\leftarrow p(x, x)$ of $\leftarrow ul(x)$ is satisfied in D and D^U, and hence that also $\leftarrow p(0, 0),\ 9999999 > th$ (the third of the simplifications in 4.2). is satisfied in D^U. However, that is wrong if $9999999 > th$ holds. Thus, uncertainty-intolerant integrity checking may wrongly infer that the high uncertainty constraint $\leftarrow uh(z)$ cannot be violated in D^U.

4.4 Uncertainty Management by Repairing (D, IC)

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

In fact, already the identification of all violations in (D, IC) may be prohibitively costly at update time. But there is only a single low uncertainty constraint violation in our example: $p(0,0)$ is the only cause of the violation $\leftarrow$ *ul(0)* in D. Thus, to begin with repairing D means to request $U =$ *delete* $p(0,0)$, and to execute U if it preserves all constraints, according to Theorem 2.

To check U for integrity preservation means to evaluate the simplifications

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

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

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

and yields the simplification $\leftarrow p(0,y),\ q(y,0),\ {\sim}p(y,0)$.

As is easily seen, this simplification is violated by each pair of facts of the form $p(0,o),\ q(o,0)$ in D, where o is an odd number in $[1, 9999999]$. Thus, deleting $q(0,0)$ for repairing the violation caused by deleting $p(0,0)$ causes the violation of each instance of the form $\leftarrow um(o)$, for each odd number o in $[1, 9999999]$.

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

4.5 Uncertainty Management by Repairing (D^U, IC)

Similar to repairing (D, IC), repairing (D^U, IC) also is more expensive than to tolerate extant constraint violations until they can be repaired at some more convenient time. That can be illustrated by the three violations in D^U, as identified in 4.1 and 4.2: the low uncertainty that already exists in D, and the medium and high uncertainties caused by U and detected by UTIC. To repair them obviously is even more intricate than to only repair the first of them as tracked in 4.4.

Moreover, for uncertainty management in safety-critical applications, it is no good idea to simply accept an update without checking for potential violations of constraints, and to attempt repairs only after the update is com-

mitted, since repairing takes time, during which an updated but unchecked state may contain possibly very dangerous uncertainty of any order.

4.6 AHI and ATU for Uncertainty Management

Checking and repairing uncertainty constraints involves their evaluation, by querying them. As already mentioned in 3.3.1, CQA is an approach to cope with constraint violations for query evaluation. However, the evaluation of constraints or simplifications thereof by CQA is unprofitable, since consistent query answers are defined to be those that are true in each minimally repaired database. Thus, for each queried denial constraint I, CQA will by definition return the empty answer, which indicates the satisfaction of I. Thus, answers to queried constraints computed by CQA have no meaningful interpretation.

For example, CQA computes the empty answer to the query $\leftarrow ul(x)$ and to $\leftarrow uh(z)$, for any extension of p and q. However, the only reasonable answers to $\leftarrow ul(x)$ and $\leftarrow uh(z)$ in D are $x = 0$ and, resp., $x = 9999999$, if $9999999 > th$. These answers correctly indicate low and high uncertainty in D and, resp., D^U.

For computing correct answers to queries (rather than to denials representing constraints), AHI and ATU are viable alternatives to CQA. A comparison which turned out to be advantageous for AHI has been presented in [18]. ATU goes beyond CQA and AHI by providing reasonable answers even if these answers depend on uncertain data that violate constraints, as we have seen in 3.3.2.

5 Scaling-Up Uncertainty Management to Concurrency

The number of concurrently issued transactions increases with the number of online users. So far, we have tacitly considered only serial executions of transactions. Such executions have many transactions wait for others to complete. Thus, the serialization of transactions severely limits the scalability of applications. Hence, to achieve high scalability, transactions should be executed concurrently, without compromising integrity, i.e., without increasing uncertainty.

Standard concurrency theory guarantees the preservation of integrity only if each transaction, when executed in isolation, translates a consistent state into a consistent successor state. More precisely, a standard result of concurrency theory says that, in a history H of concurrently executed transactions $T_1, \ldots, T_n$, each T_i preserves integrity if it preserves integrity when executed non-concurrently and H is serializable, i.e., the effects of the transactions

in H are equivalent to the effects of a serial execution of $\{T_1, \ldots, T_n\}$. For convenience, let us capture this result by the following schematic rule:

isolated integrity + serializability $\Rightarrow$ *concurrent integrity* (*)

Now, if uncertainty corresponds to integrity violation, and each transaction is supposed to operate on a consistent input state, then (*) does not guarante that concurrently executed transactions on uncertain data would keep uncertainty at bay, even if they would not increase uncertaity when executed in isolation and the history of their execution was serializable.

Fortunately, however, the approaches and results in Section 3 straightforwardly scale up to concurrent transactions without further ado, as shown for inconsistency-tolerant integrity checking in [23], based on a mesure that compares sets of violated instances of constraints before and after a transaction.

Theorem 3 below adapts Theorem 3 in [23] to measure-based UTIC in general. It asserts that a transaction T in a history H of concurrently executing transactions does not increase uncertainty if H is serializable and T preserves integrity whenever it is executed in isolation. On one hand, Theorem 3 weakens Theorem 3 [23] by assuming strict two-phase locking (abbr. S2PL), rather than abstracting away from any implementation of serializability. On the other hand, Theorem 3 generalizes Theorem 3 [23] by using an arbitrary uncertainty measure μ, rather than the inconsistency measure mentioned above. A full-fledged generalization that would not assume any particular realization of serializability is possible along the lines of [23], but would be out of proportion in this paper.

Theorem 3. Let H be a S2PL history, μ an uncertainty measure and T a transaction in H that uses a μ-based UTIC method for checking the integrity preservation of its write operations. Further, let D be the committed state at which T begins in H, and D^T the committed state at which T ends in H. Then, $\mu(D, IC) \preccurlyeq \mu(D^T, IC)$.

The essential difference between (*) and Theorem 3 is that the latter is uncertainty-tolerant, the former is not. Thus, as opposed to (*), Theorem 3 identifies useful sufficient conditions for integrity preservation in the presence of uncertain data. Another important difference is that the guarantees of integrity preservation that (*) can make for T require the integrity preservation of all other transactions that may happen to be executed concurrently with T. As opposed to that, Theorem 3 does away with the standard premise of (*) that all transactions in H must preserve integrity in isolation; only T itself is required to have that property. Thus, the guarantees that Theorem 3 can make for individual transactions T are much better than those of (*).

To outline a proof of Theorem 3, we distinguish the cases that T either terminates by aborting or by committing its write operations. If T aborts, then Theorem 3 holds vacuously, since, by definition, no aborted transaction could have any effect whatsoever on any committed state. So, we can suppose

that T commits. Let $\mathcal{M}$ be the μ-based method used by T. Since T commits, it follows that $\mathcal{M}(D, IC, W_T) = true$, where W_T is the write set of T, i.e., $D^T = D^{W_T}$, since otherwise, the writes of T would violate integrity and thus T would abort. Since H is S2PL, it follows that there is an equivalent serialization H' of H that preserves the order of commited states in H. Thus, D and D^T are also the committed states at beginning and end of T in H'. Hence, Theorem 3 follows from $\mathcal{M}(D, IC, W_T) = true$ and Definition 2 since H' is serial, i.e., non-concurrent.

It follows from Theorem 2 that, similar to UTIC, also integrity repairing scales up to S2PL concurrency if realized as described in 3.2, i.e., if UTIC is used to check candidate repairs for integrity preservation.

Also AHI and ATU as defined in 3.3 scale up to concurrency, which can be seen as follows. Concurrent query answering is realized by read-only transactions. In S2PL histories, such transactions always read from committed states that are identical to states in equivalent serial histories, as described in the proof of Theorem 3. Hence, each answer can be checked for certainty or for being within in the confines of tolerable uncertainty as described in 3.3.1 or, resp., 3.3.2.

6 A Reference Architecture for Uncertain Database Management under Concurrent Transactions

Figure 1 shows the reference architecture for uncertain database management under concurrent transactions, integrated with a classical DBMS architecture, which is the main result of our research. Here, a *modular* architecture is depicted.

The modules refereed in Figure 1 are the following (from the bottom to up):

- *Physical Layer*: it is the physical layer of classical DBMS, where data are stored according to some *storage scheme* (e.g., based on fixed record length);
- *Access Methods*: it contains the collection of access methods needed to retrieve data from the physical layer as a reaction to the execution of standard SQL statements that occur in the relational layer;
- *Relational Layer*: it is the relational layer of classical DBMS, where tuples are processed according to the relational data model;
- *Uncertainty Detection Layer*: it is the layer where data uncertainty are detected, according to a given uncertain data model (e.g., *probabilistic uncertain model*);
- *Uncertainty Management Layer*: it is the layer where data uncertainty is managed by using the elements (denoted by $u\_mi(.,.,.)$) of an integrated

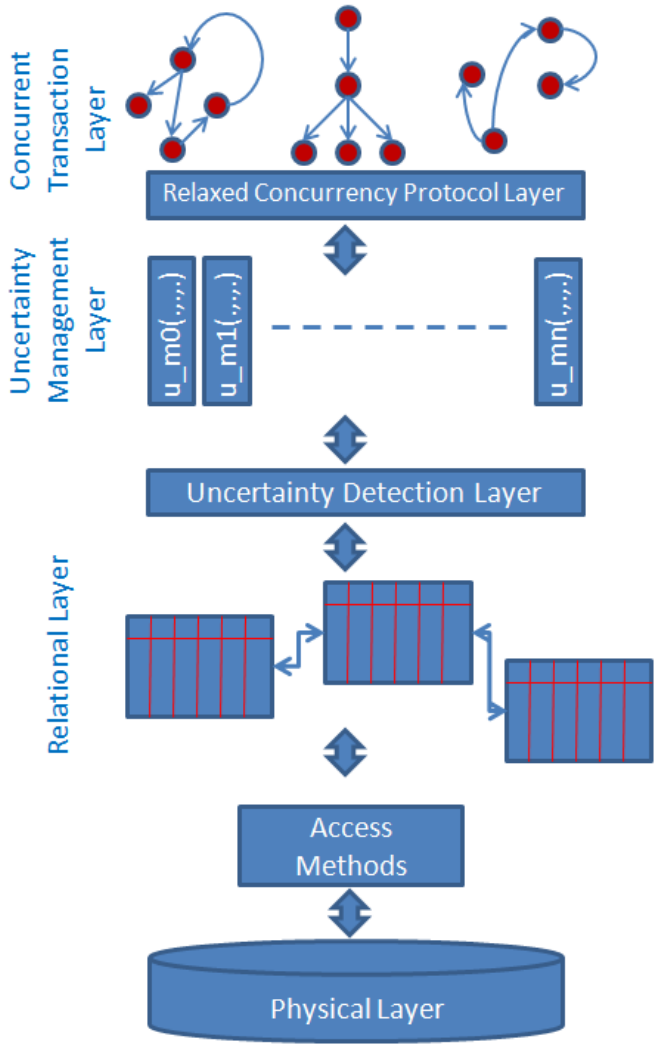

Fig. 1 Reference architecture for uncertain database management under concurrent transactions

approach to uncertainty management, such as a combination of UTIC and ATU, as proposed in Section 3 and Section 4;

- *Relaxed Concurrency Protocol Layer*: where concurrency is handled in an inconsistency-tolerant manner, as proposed in Section 3 and Section 4, such that sequentializable histories of concurrent transactions can proceed without rollbacks that would be due to extant uncertainty;

- *Concurrent Transaction Layer*: it is the layer where concurrent transactions (modeled in terms of *direct graphs*) occur, for instance in the application scenario of an *e*-commerce Web database system.

7 Related Work

An early, not yet measure-based attempt to conceptualize some of the material in 3.1 has been made in [21]. Apart from that, it seems that integrity maintenance and query answering in the presence of uncertain data never have been approached in a uniform way, as in this paper. That is surprising since integrity, uncertainty and answering queries with certainty are obviously related.

Semantic similarities and differences between uncertainty and the lack of integrity are observed in [29]. In that book, largely diverse proposals to handle data that suffer from uncertainty are discussed. In particular, approaches such as probabilistic and fuzzy set modeling, exception handling, repairing and paraconsistent reasoning are discussed. However, no particular approach to integrity maintenance (checking or repairing) is considered. Also, no attention is paid to concurrency.

Several paraconsistent logics that tolerate inconsistency and thus uncertainty of data have been proposed, e.g., in [7, 9]. Each of them departs from classical first-order logic, by adopting some annotated, probabilistic, modal or multivalued logic, or by replacing standard axioms and inference rules with non-standard axiomatizations. As opposed to that, UTIC fully conforms with standard datalog and does not need any extension of classical logic.

Work concerned with semantic inconsistencies in databases is also going on in the field of measuring inconsistency [27]. However, the violation measures on which UTIC is based have been conceived to work well also in databases with non-monotonic negation, whereas the inconsistency measures in the literature do not scale up to non-monotonicity, as argued in [20].

8 Conclusions and Future Work

We have applied and extended recently developed concepts of logical inconsistency tolerance to problems of managing uncertainty in databases.

We have shown that the uncertainty of stored data can be modeled by integrity constraints and maintained by uncertainty-tolerant integrity management technology. In particular, updates can be monitored by UTIC, such that they do not increase uncertainty, and extant uncertainty can be partially repaired while tolerating remaining uncertainty. Also, we have outlined how databases can provide reasonable answers in the presence of uncertainty.

Moreover, we have highlighted that uncertainty tolerance is necessary and sufficient for scaling up uncertainty management in databases to concurrent transactions. This result is significant since concurrency is a common and indeed indispensable feature of customary database management systems.

As illustrated in Section 4, the use of uncertainty-tolerant tools is essential, since wrong, possibly fatal conclusions can be inferred from deficient data by using a method that is uncertainty-intolerant. A lot of UTIC methods, and some intolerant ones, have been identified in [22, 19].

In ongoing research, we are elaborating a generalization of the results in Section 5 to arbitrarily serializable histories. Also, we are working on scaling up our results further to replicated databases and recoverable histories. Moreover, we envisage further applications of inconsistency-tolerant uncertainty management in the fields of OLAP, data mining, and data stream query processing, in order to complement our work in [16, 13, 14, 15, 8].

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. Afrati and P. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *Proc. 12th ICDT*, pages 31–41. ACM Press, 2009.
3. M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of PODS*, pages 68–79. ACM Press, 1999.
4. H. Bauer. *Maß- und Integrationstheorie*. De Gruyter, 2nd edition edition, 1992.
5. J. Berlin and A. Motro. Tuplerank: Ranking discovered content in virtual databases. In *6th NGITS*, volume 4032 of *LNCS*, pages 13–25. Springer, 2006.
6. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
7. L. Bertossi, A. Hunter, and T. Schaub. *Inconsistency Tolerance*, volume 3300 of *LNCS*. Springer, 2005.
8. B. P. Budhia, A. Cuzzocrea, and C. K.-S. Leung. Vertical frequent pattern mining from uncertain data. In *KES*, pages 1273–1282, 2012.
9. W. Carnielli, M. Coniglio, and I. D'Ottaviano, editors. *The Many Sides of Logic*, volume 21 of *Studies in Logic*. College Publications, London, 2009.
10. S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 254–262. Morgan Kaufmann, 2000.
11. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. on Database Syst. (TODS)*, 15(2):162–207, 1990.
12. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
13. A. Cuzzocrea. Olap over uncertain and imprecise data: Fundamental issues and novel research perspectives. In *Proc. 21st DEXA Workshop*, pages 331–336. IEEE CSP, 2001.
14. A. Cuzzocrea. Retrieving accurate estimates to olap queries over uncertain and imprecise multidimensional data streams. In *Proc. 23rd Scientific and Statistical Database Management*, volume 6809 of *LNCS*, pages 575–576. Springer, 2011.

15. A. Cuzzocrea and H. Decker. Non-linear data stream compression: Foundations and theoretical results. In *Proc. 7th HAIS*, volume 7208 of *LNCS*, pages 622–634. Springer, 2012.
16. A. Cuzzocrea and D. Gunopulos. Efficiently computing and querying multidimensional olap data cubes over probabilistic relational data. In *ADBIS*, pages 132–148, 2010.
17. H. Decker. The range form of databases and queries or: How to avoid floundering. In *Proc. 5th ÖGAI*, volume 208 of *Informatik-Fachberichte*, pages 114–123. Springer, 1989.
18. H. Decker. Answers that have integrity. In K.-D. Schewe and B. Thalheim, editors, *Semantics in Data and Knowledge Bases - 4th International Workshop SDKB*, volume 6834 of *LNCS*, pages 54–72. Springer, 2011.
19. H. Decker. Inconsistency-tolerant integrity checking based on inconsistency metrics. In *Proc. KES, Part II*, volume 6882 of *LNCS*, pages 548–558. Springer, 2011.
20. H. Decker. Measure-based inconsistency-tolerant maintenance of database integrity. In *Semantics in Data and Knowledge Bases - 5th International Workshop SDKB*, volume 7693 of *LNCS*, pages 149–173. Springer, 2013.
21. H. Decker and D. Martinenghi. Integrity checking for uncertain data. In *Proc. 2nd TDM Workshop on Uncertainty in Databases*, volume WP06-01 of *CTIT Workshop Proceedings Series*, pages 41–48, The Netherlands, 2006. Univ. Twente.
22. H. Decker and D. Martinenghi. Inconsistency-tolerant integrity checking. *Transactions on Knowledge and Data Engineering*, 23(2):218–234, 2011.
23. H. Decker and F. D. M. noz Escoí. Revisiting and improving a result on integrity preservation by concurrent transactions. In *Proc. OTM Workshops*, volume 6428 of *LNCS*, pages 297–306. Springer, 2010.
24. T. H. Fung and R. Kowalski. The iff proof procedure for abductive logic programming. *J. Logic Programming*, 33(2):151–165, 1997.
25. J. Grant and A. Hunteri. Measuring the good and the bad in inconsistent information. In *Proc. 22nd IJCAI*, pages 2632–2637, 2011.
26. A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *Proceedings of PODS 1994*, pages 45–55. ACM Press, 1994.
27. A. Hunter and S. Konieczny. Approaches to measuring inconsistent information. In *Inconsistency Tolerance*, volume 3300 of *LNCS*, pages 191–236. Springer, 2005.
28. S. Y. Lee and T. W. Ling. Further improvements on integrity constraint checking for stratifiable deductive databases. In *VLDB'96*, pages 495–505. Kaufmann, 1996.
29. A. Motro and P. Smets. *Uncertainty Management in Information Systems: From Needs to Solutions*. Kluwer, 1996.
30. J. Wijsen. Database repairing using updates. *Transaction on Database Systems*, 30(3):722–768, 2005.