Abstract

Business rules are a cornerstone of the consistency management of e-commerce data. The concurrency of transactions by multiple customers of e-commerce services (e.g., seat reservation systems) is indispensable for most of such services. However, the enforcement of business rules may falter due to extant inconsistencies of legacy data and integrity violations caused by the concurrency of transactions. To avoid this, we propose an inconsistency-tolerant approach to business rules enforcement for e-business applications in which business rules are expressed by database integrity constraints. The extension overcomes problems related to persistent integrity violations of stored data in e-commerce systems with concurrent transactions.

1 Introduction

Business rules, also known as integrity constraints, are capable of improving the quality of e-commerce data and processes significantly, as pointed out, e.g., in [6, 22].

Unfortunately, database vendors rarely encourage an implementation of business rules by declarative integrity constraints. Their products only offer quite modest constructs for a declarative description of the semantics of stored data. That can be blamed on the circumstance that theoretical solutions are not considered to be flexible enough for working well in practice.

Theoretical solutions for enforcing declaratively stated integrity constraints are broadly documented in the literature (for surveys, see [7] [18]).

The practical use of declarative business rules for integrity management is described, e.g., in [19] [23] [24]. A vast literature also exists for managing concurrent transactions [11] [13] [3] [15].

This paper deals with two problems that impede a combination of the identified separate lines of research, for enabling the preservation of the integrity of stored data for concurrent e-commerce transactions. Each of the two problems can be identified with a common tacit assumption. We are going to see that both assumptions are hardly more than wishful thinking in practice.

One of these assumptions is that an update can be efficiently checked for integrity only if the state before the update totally satisfies all constraints without exception. We call this assumption the total integrity premise. The other is that, for guaranteeing integrity preservation by concurrent transactions, each transaction is supposed to preserve integrity when executed in isolation. We call this assumption the isolated integrity premise.

The dispensability of the total integrity premise has been discovered in [8]. However, the isolation of transactions for avoiding anomalies of concurrency caused by extant constraint violations has not yet been taken into account in the literature.

In section 2, we characterize the premises of total and isolated integrity. In section 3, we recapture the concept of inconsistency-tolerant integrity checking. We are going to see that it serves to waive the total integrity premise as well as to relax the isolated integrity premise. In section 4, we address related work, with attention to the literature on integrity checking for concurrent transactions. Unless specified otherwise, we use terminology and notations that are conventional in the database community [22].
2 Problems

The premises of total and isolated integrity are explained in 2.1 and 2.2, respectively.

2.1 The Total Integrity Premise

Integrity checking can be prohibitively costly, unless some simplification mechanisms are used [5]. That can be illustrated as follows. (As usual, lower-case letters \(x, y, z\) denote variables, in the example below.)

Example 1

Let

\[
I = \lnot \text{proj}(x, y), \text{proj}(x, z), y \neq z
\]

be a primary key constraint on the first column of relation \(\text{proj}\), which stores information about projects: the first column be the project identifier. Further, let \(\text{emp}\) be a relation about employees: the first column be the employee’s name and the second a project to which s/he is assigned. The foreign key constraint

\[
I' = \forall x, y \exists z (\text{emp}(x, y) \rightarrow \text{proj}(y, z))
\]

on the second column of \(\text{emp}\) references the primary key of \(\text{proj}\). Further, let a transaction consists of inserting \(\text{emp}(\text{Fred}, p)\). Most integrity checking methods \(M\) ignore \(I'\) because it does not constrain the relation \(\text{emp}\). Rather, they only evaluate the instance (later called ‘case’)

\[
\exists z (\text{emp}(\text{Fred}, p) \rightarrow \text{proj}(p, z))
\]

of \(I\), or rather its simplification

\[
\exists z \text{proj}(p, z)
\]

since \(\text{emp}(\text{Fred}, p)\) becomes true by the transaction. If, e.g., \((p, e)\) is a row in \(\text{proj}\), \(M\) sanctions the insertion. If there is no tuple matching \((p, z)\) in \(\text{proj}\), then \(M\) signals a violation of integrity.

The correctness proofs of methods for simplified integrity checking in the literature all rely on the total integrity premise, i.e., that integrity always be totally satisfied, before updates are checked for preserving consistency. In practice, however, it is rather the exception than the rule that this premise is fulfilled. In particular for applications such as e-commerce, 24/7 services, distributed and replicated databases, legacy data maintenance, data warehousing, data federation, etc, a certain amount of extant integrity violations in committed states has to be lived with, at least temporarily.

Working around or repairing such inconsistencies on the spot may unduly disturb running operations. Repairing may also be out of reach because many inconsistencies tend to be hidden and overlooked. They may remain undetected for extended periods of time. Hence, the total integrity premise, which traditionally is assumed unanimously, does not approve the correctness of integrity checking in practice, since it often is performed in the presence of extant consistency violations. Fortunately, however, that premise can be waived without incurring any cost and without losing its essential guarantees, as shown in 3.1.

2.2 The Isolated Integrity Premise

Integrity preservation has been a key issue in the literature on transaction processing since its beginnings. We cite from [11]: “it is assumed that each transaction, when executed alone, transforms a consistent state into a consistent state; that is, transactions preserve consistency”. This is what we have called, in the introduction, the isolated integrity premise. (We recall that the execution of a transaction \(T\) is isolated when it is not concurrent with other transactions, or when the state transition effected by \(T\) is as if having been executed alone.) From that premise, the well-known result is inferred that then, also all sequentializable schedules of concurrent transactions preserve ‘consistency’, i.e., integrity.

Similar to the total integrity premise, to assume the isolated integrity premise seems to be illusionary, in general, and in particular for distributed multi-user databases. In fact, it is hard to believe than any (human or programmed) client who issues a transaction \(T\) would ever bet on a consistency-preserving outcome of \(T\) by blindly trusting that all other clients have taken the same care as herself for making sure that their (con-
current) transactions preserve integrity in isolation. Still, in practice, most clients are confident about the integrity of the outcome of their transactions, although there is no theory to justify their optimism. Such a justification is given in 3.1. In 3.2, we show that the consistency guarantees of inconsistency-tolerant integrity checking can be extended to concurrent transactions.

3 Inconsistency Tolerance

For enterprise computing, the purpose of business rules is to state and enforce semantic integrity properties of e-commerce data. However, inconsistencies are unavoidable in practice. Rather than insisting that all business rules must be totally satisfied at all times, it is necessary to tolerate unavoidable integrity violations, sometimes.

Whenever time permits, attempts of reducing or repairing such inconsistencies can be made, while such attempts often are unaffordable at update time. Thus, updates should be checkable for integrity preservation, even if there are extant integrity violations, which can be dealt with later. That is the philosophy behind inconsistency-tolerant integrity checking, as revisited in 3.1. In 3.2, we outline a generalization of the results in 3.1 to concurrent transactions.

Throughout the rest of the paper, let the symbols $D$, $I$, $IC$, $T$, $M$ stand for a database, an integrity constraint, a set of integrity constraints, a transaction and, resp., an integrity checking method. We suppose that all constraints are represented in prenex form, i.e., all quantifiers of variables appear leftmost. That includes the two most common forms of representing integrity constraints: as denials or in prenex normal form.

In general, each method $M$ can be conceived as a mapping which takes triples $(D, IC, T)$ as input, and outputs either $OK$, which means that $M$ sanctions $T$ as integrity-preserving, or $KO$, which indicates that executing $T$ would violate some constraint. Further, let $D^T$ denote the database state obtained by applying the writeset of $T$ to $D$.

3.1 Waiving the Total Integrity Premise

In [8], it is shown that, contrary to common belief, it is possible to waive the total integrity premise for most simplification-based approaches to integrity checking without any trade-off. Methods which continue to function well when this premise is waived are called inconsistency-tolerant. The basic idea is illustrated below.

Example 2

Let $I$ and $I'$ be as in Example 1. For checking if inserting $(Fred, p)$ preserves integrity, most integrity checking methods $M$ sanction this update if, e.g., $(p, e)$ is a row in $proj$. Now, the positive outcome of this integrity check is not disturbed if, e.g., also the tuple $(p, f)$ is a row in $proj$. This may at first be irritating, since $I$ then is violated by two tuples about project $p$ in $proj$. In fact,

$$\leftarrow proj(p, e), proj(p, f), e \neq f$$

indicates an integrity violation. However, this violation has not been caused by the insertion just checked. It has been there before, and the assignment of $Fred$ to $p$ should not be rejected just because the data about $p$ are not consistent. After all, it may be part of $Fred$’s new job to cleanse potentially inconsistent project data. In general, a transaction $T$ that preserves the integrity of all data should not be rejected. And that is precisely what $M$’s output indicates: no instance of any constraint that is satisfied in the state before $T$ is committed is violated after $T$ has been committed.

The concept of inconsistency-tolerant integrity checking [8] is formalized as follows.

Definition 1 (inconsistency tolerance)

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

b) A constraint $I'$ obtained by a substitution of the global variables in $I$ is called a case of $I$.

c) Let $\text{SatCas}(D, IC)$ be the set of all cases $C$ of constraints in $IC$ such that $C$ is satisfied in $D$.

d) $M$ is called inconsistency-tolerant if, for each triple $(D, IC, T)$, and each $C \in \text{SatCas}(D, IC)$, (*) holds:

$$M(D, IC, T) = OK \Leftrightarrow C \text{ is satisfied in } D^U \quad (*)$$
In example 1, the global variables of \( I' \) are \( x \) and \( y \); all variables of \( I \) are global. As we have already seen, only a simplification of the case
\[
\exists z (Fred, p) \rightarrow \text{proj}(p, z)
\]
of \( I' \) is checked by most methods. All irrelevant cases, e.g., the violated case
\[
\leftarrow \text{proj}(p, e), \text{proj}(p, f), e \neq f
\]
of \( I \), are ignored and thus tolerated.

It is easy to see that the above definition generalizes the traditional definition of sound and complete integrity checking. The essential difference is that, traditionally, the total integrity premise is imposed, so that, for an integrity checking method \( \mathcal{M} \) to be correct, (*) is required for each \( I \) in \( IC \), not just for the cases \( C \) that are satisfied in the state before the update. Thus, \( \mathcal{M} \) does not worry about extant constraint violations.

In other words, a method \( \mathcal{M} \) is inconsistency-tolerant if its output \( OK \) for a given transaction \( T \) guarantees that all instances of constraints that are satisfied in the before-state of \( T \) will remain satisfied after \( T \) has been committed and executed. However, each transaction that, on purpose or by happenstance, repairs some inconsistent instance(s) of any constraint without introducing any new violation will be OK-ed too by \( \mathcal{M} \). This means that, over time, the amount of integrity violations will decrease, as long as an inconsistency-tolerant method is used for checking each transaction for integrity preservation.

Note that it follows by the definition above that each inconsistency-tolerant \( \mathcal{M} \) outputs \( KO \) for any transaction the commitment of which would violate a hitherto satisfied instance of some constraint. It is then up to the agent who has called \( \mathcal{M} \) for checking integrity to react appropriately to the output \( KO \).

A conservative reaction is to simply cancel and reject the transaction. A more constructive reaction could be to automatically modify the transaction so that its execution preserves integrity (e.g., by cascading deletes), or to obtain such a modification via a dialogue with the agent who has issued the transaction (e.g., by an ask-the-user facility). In this paper, we do not deal with such options.

It has been shown in [8] and [9] that many (though not all) integrity checking methods are inconsistency-tolerant. Interestingly, also the behavior of built-in deferred integrity constraint checking in DBMSs on the market is inconsistency-tolerant. Hence, the cited results justify the use of such methods in systems where extant violations have to be lived with. Replicated databases are an example of such systems.

### 3.2 Relaxing the Isolated Integrity Premise

To say, as the isolated integrity premise does, that a transaction \( T \) “preserves integrity in isolation”, means the following: For a given set \( IC \) of integrity constraints and each state \( D \) of a given database schema, each \( I \in IC \) is satisfied in \( DT \) if \( I \) is satisfied in \( D \).

Now, our objective is to apply the concept of inconsistency-tolerant integrity checking in 3.1 not only to transactions executed in isolation, but also to concurrent transactions. In order to do so, let us restrict, for simplicity, the isolated integrity premise and its conclusion to a single but arbitrarily complex integrity constraint \( I \).

The isolated integrity premise (cf. 2.2) then guarantees that, for each state \( D \) of a given database schema such that \( I \) is satisfied in \( D \), and for each transaction \( T \) that preserves \( I \) in isolation, \( I \) will remain satisfied in \( DT \) if \( T \) and all transactions that are concurrent with \( T \) are sequentializable and preserve \( I \) in isolation.

Since each case of each constraint is itself a constraint, it follows that the preceding result also holds for each case \( C \) that is satisfied in \( D \). Now, recall that the condition (*) of the definition of inconsistency tolerance in 2.1 holds for each case of \( IC \) that is satisfied in \( D \). So, we can conclude that each inconsistency-tolerant method can be used for integrity checking also when transactions are concurrent.

More precisely, we can state the following result: For each state \( D \) of a database on which the set of integrity constraints \( IC \) is imposed, and for each transaction \( T \) that preserves some case \( C \) of some constraint in \( IC \) in isolation, \( C \) will remain satisfied in \( DT \) if all transactions before \( T \) or concurrent with \( T \) are sequentializable and also preserve \( C \) in isolation.
Note that this result does not mean that each case would have to be checked individually. On the contrary: integrity checking can proceed as for systems without concurrency, i.e., no built-in nor any external routine that takes part in the integrity checking process needs to be modified. The result just says that, if the method outputs OK, then everything that was satisfied in the state before the transaction will remain satisfied in the after the transaction has committed, also for concurrent e-business transactions.

The essential difference of this relaxation with regard to the traditional result, which imposes the general isolated integrity premise, is the following. In the relaxed result, isolated integrity preservation only is asked to hold for individual cases. Since simplified integrity checking always focuses on cases that are relevant for the writeset of a given transaction \( T \), only these cases are guaranteed to remain satisfied by a successful integrity check. All non-relevant cases of the same or any other constraints may possibly be violated by concurrent or preceding transactions. Such violations are detected only if the respective transactions are checked too. If not, such violations are tolerated by each inconsistency-tolerant method that is used to check \( T \).

For fairness, the following caveat must be mentioned. The relaxation of the isolated integrity premise outlined above still asks for an unrestricted isolation level with regard to individual cases. This means that we cannot expect the generalized result to hold if the isolation level is lowered. (For a general critique of lowering isolation levels, see [2].)

### 4 Related work

As already indicated in the introduction, most papers that study methods for checking the preservation of integrity do not deal with transaction concurrency. On the other hand, most papers that discuss concurrent transactions take it for granted that, if transaction were checked for integrity preservation in isolation, then it would pass that test successfully, i.e., they do not care how integrity is or would be checked.

In early work [14, 10, 16, 11, 1, 12], a distinction is made between integrity violations caused either by anomalies of concurrency or by semantic errors. In [10, 16], concurrency is not dealt with any further. In [14, 11, 1, 12], integrity is not looked at in detail. Also in later related work, either concurrency or integrity is largely passed by, except in [4] [17], to be addressed further down, and in papers (e.g., [25]) that do not check each transaction for integrity, but only the final state reached at the end of a history of concurrent transactions.

In this paper, we have not dealt with final-state integrity because we are interested in the integrity of each committed state, i.e, in guarantees that can be made for each individual transaction. Moreover, for each serializable history \( H \), integrity is preserved in the final state of \( H \) if each individual transactions in \( H \) preserves integrity, even if there are extant constraint violations.

In [4], Böttcher observes that integrity checks are read-only actions without effect on other operations, possibly except abortions due to integrity violation. Some scheduling optimizations made possible by the unobtrusive nature of read actions for integrity checking are discussed in [4].

A different, more DBMS-oriented solution, which however is proprietary, has been proposed in [21]:

“Making constraints immediate at the end of a transaction is a way of checking whether COMMIT can succeed. You can avoid unexpected rollbacks by setting constraints to IMMEDIATE as the last statement in a transaction. If any constraint fails the check, you can then correct the error before committing the transaction.”

However, as long as the semantics of IMMEDIATE are not well-defined, it seems to be difficult to identify against which state a constraint is evaluated. In particular, the meaning of IMMEDIATE seems to be entirely speculative if there are locks which prevent immediate access (whatever that may mean).

A non-proprietary and partially automatic way to re-program concurrent transactions such that unwanted conflicts at commit time are avoided is proposed in [17]. In that paper, the authors describe how to augment transactions with read
actions for simplified integrity checking and with locks, so that their serializable execution guarantees integrity preservation. Ad-hoc transactions are not considered in [17].

As opposed to solutions that are proprietary or oriented toward transaction design, we advocate a different approach: For each transaction $T$, the DBMS should determine autonomously (either by using a built-in procedure or some external device) whether the state transition effected by $T$ preserves integrity, and react accordingly. In this paper, we have proposed ways to overcome some of the obstacles that hitherto may have impeded researchers and developers to strive for such a solution.

For replicated database systems, the interplay of built-in integrity checking, concurrency and replication consistency has been studied in [20]. In that paper, solutions are provided for enabling integrity checking even in systems where the isolation level of transactions is lowered to snapshot isolation [2]. However, inconsistency tolerance in the sense of coping with extant integrity violations has not been considered in [20]. Thus, for the snapshot-isolation-based replication of databases, more research is necessary in order to clarify which consistency guarantees can be given when inconsistency-tolerant integrity checking methods are used in the presence of inconsistent cases of constraints.

## 5 Conclusion

For concurrent transactions, the onus of maintaining the integrity of business rules has, up to now, been on the designers and users of applications. We expect that, in the long run, this will give way to specifications of integrity constraints that can be supported automatically, just the way they are supported already in centralized, non-distributed database systems.

Thus, a guideline for our work on augmenting e-business transactions with business rules has been to be as declarative as possible. The advantage of declarativity is to free users and application programmers from having to worry about integrity preservation. That is, the database designer should state business rules as declarative integrity constraints in SQL and leave everything else to the integrity checking module of the DBMS. That module may be built into the DBMS core or run on top of it. In any case, the enforcement of the integrity of business rules should be as transparent to the user as concurrency, distribution and replication.

However, as we have seen, established authors of concurrency theory take it easy by requiring the near-impossible: that all issued transaction should be programmed such that they can guarantee to preserve integrity in isolation [13] [3]. So, database designers and users feel compelled to program transactions in a way such that as much unwanted situations as possible are avoided. All related work we have found in the literature tends to be of that kind.

Thus, the goal which has motivated this paper has been to make the enforcement of business rules feasible for concurrent e-business transactions. We have identified two obstacles that have hitherto prevented to reach that goal: the requirements of the premises of total and isolated integrity.

For overcoming the traditional belief that integrity can be checked efficiently for a given transaction $T$ only if the state before $T$ is totally satisfied, we have revisited the work in [8]. There, it has been shown that the total integrity premise can simply be waived without problems, for most (though not all) integrity checking methods. Fortunately, the premise also is unnecessary for deferred integrity checking of key constraints and other common built-in integrity constructs in DBMSs on the market.

We have seen that the advantages of making the total integrity premise obsolete even extend to relaxing the isolated integrity premise. More precisely, the use of an inconsistency-tolerant integrity checking method to enforce business rules for concurrent sequentializable transactions guarantees that no transaction can violate any instance of any constraint that has been satisfied in the state before committing if all transactions preserve the integrity of the same instance in isolation. Conversely stated, our result guarantees that, if any violation happens, then no transaction that
has been correctly and successfully checked for integrity preservation by an inconsistency-tolerant method can be held responsible for that. The most interesting aspect of this result is that it even holds in the presence of extant inconsistencies.

We have seen that more research is needed for systems in which the isolation level of concurrent transactions is compromised. In particular, for non-sequentializable histories of concurrent transactions, it should be interesting to elaborate a precise theory of different kinds of database states. Such a theory should allow to differentiate between states that are committed, states that are “seen” by a transaction and states that are “seen” by (human or programmed) agents that have issued the transaction, and which consistency guarantees can be made by which methods for transitions between those states. This area of research is important because most commercial database management systems compromise the isolation level of transactions in favor of a higher transaction throughput, while leaving the problem of integrity preservation to the application programmers.

References


