BUSINESS RULES FOR REPLICATED ENTERPRISE DATA

Hendrik Decker and Francesc Muñoz-Escoí
Instituto Tecnológico de Informática, CiudadPolitécnica de la Innovación
Universidad Politécnica de Valencia, Campus de Vera 8G, 46071 Valencia, Spain

ABSTRACT

Business rules are essential for the IT alignment of enterprises. Their enforcement may conflict with the commit guarantees of replication protocols. Business rule enforcement may also falter due to integrity violations caused by concurrent transactions. To avoid this, we propose an inconsistency-tolerant extension of replication architectures for data-centric enterprise applications that supports business rules expressed by database integrity constraints.

KEYWORDS: Business rules, database integrity, concurrent transactions

1 INTRODUCTION

Data replication boosts the availability, reliability and fault tolerance of business-critical applications (Wiesman et al, 2005; Armendáriz et al, 2007). Business rules, also known as integrity constraints, improve the quality of business data and processes (Date, 2000; Doorn et al, 2002; Ross, 2003; Ramakrishnan et al, 2006). Thus, a combination of replication and business rules promises a considerable synergy effect.

This paper deals with three problems that impede a combination of replication and business rules. First, replication protocols may conflict with an atomicity-preserving evaluation of integrity constraints. We call this *the replication-integrity conflict*. The second and third problems are two common tacit assumptions for integrity checking. One is that integrity can be efficiently checked 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*. In section 2, we characterize the replication-integrity conflict and the premises of total and isolated integrity. In section 3, we propose extensions of protocols for avoiding replication-integrity conflicts. In section 4, we show that inconsistency-tolerant integrity checking serves to waive the total integrity premise and to relax the isolated integrity premise. We use conventional terminology and notations (Ramakrishnan et al, 2003).

2 PROBLEMS

2.1 The replication-integrity conflict

Most DBMSs offer two modes of integrity checking: *immediate* or *deferred*. In immediate mode, each action in a transaction *T* is immediately checked for integrity preservation. *T* continues if integrity remains satisfied, else *T* aborts. In deferred mode, integrity is checked only after *T* has requested its commit. That request then is complied with if and only if the integrity check has detected no violation.

Each transaction atomically maps the *before*-state, in which the transaction is issued, to an *after*-state, obtained by executing all actions of the transaction (Gray,1981). Clearly, integrity checking in immediate mode destroys atomicity. Of course, some specific applications may want to have exactly the effects of immediate checking. But, since we are interested in reliable results of integrity checking in the general case, we do not consider the immediate mode of integrity checking any further.

Existing replication protocols do not take care for integrity checking. In replicated multi-master databases, concurrent transactions may start and execute in different nodes, and proceed without problem until they request commitment. (If, for some reason, a transaction T needs to abort, transaction handling can be assumed to undo all uncommitted changes caused by T, so as if they never have happened.) Upon receipt of the commit request of a transaction T, the replication protocol validates T and guarantees its commit if there is no read-write or write-write conflict among concurrent transactions.

Now, an integrity conflict may arise if constraints are checked in deferred mode, after successful conflict validation by the replication protocol. Thus, an integrity violation may be diagnosed that remained unnoticed by the replication protocol. So, a transaction that has already committed may have to be undone later, by some compensating transaction. That, however, severely compromises durability, i.e., one of the basic properties required from committed transactions (Gray, 1981).

Example 1
Let $I = \leftarrow proj(x,y), proj(x,z), y \neq z$ be a primary key constraint, represented in denial form, on the first column of relation *proj*: the first column be the project identifier and the second a vector of project attributes. Further, in relation *emp*, the first column be the employee's name and the second a project to which s/he is assigned. The foreign key constraint $I' = \forall x, y\ \exists\ z\ (emp(x,y) \rightarrow proj(y,z))$ on the second column of *emp* references the primary key of *proj*. Further, let $T = \{insert\ emp(Fred, p), delete\ proj(p, e)\}$ be a transaction that assigns *Fred* to project p and deletes the record of p. Clearly, T violates I'. But the validation of T by the replication protocol will not notice that. Thus, T may commit before its integrity violation is noticed. □

In section 3, we analyze popular classes of replication protocols and propose modifications thereof such that replication-integrity conflicts as in example 1 are avoided.

2.2 The total integrity premise

Integrity checking can be prohibitively costly, unless some simplification mechanisms are used (Christiansen et al, 2006). That can be illustrated as follows.

Example 2
Let the constraints I and I' be as in example 1. Further, let a transaction consists of inserting *emp*(*Fred*, *p*). Most integrity checking methods M ignore I' because it does not constrain the relation *emp*. Rather, they only evaluate the case $\exists$ z emp(Fred, p) $\rightarrow proj(p, z))$ of I', or rather its simplification $\exists z\ proj(p, z)$, since *emp*(*Fred*, *p*) becomes *true* by the transaction. If, e.g., (p, e) is a row in *proj*, M sanctions the insertion. If there is no tuple matching (p, z) in *proj*, then M signals a violation of integrity. □

The correctness proofs of methods for simplification-based integrity checking in the literature all rely on the premise that integrity always be totally satisfied, before updates are admitted and checked for preserving consistency. That is the *total integrity premise*. In practice, however, it is rather the exception than the rule that this premise is fulfilled. In particular for applications such as legacy data maintenance, data warehousing, data federation, 24/7 services and also in distributed and replicated databases, a certain amount of extant integrity violations 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 inconsistencies are hidden and overlooked. Hence, the total integrity premise, traditionally assumed unanimously, does not approve the correctness of integrity checking in practice, if it is performed in the presence of extant consistency violations. Fortunately, that premise can be waived without incurring any cost and without losing its essential guarantees, as shown in section 4.

2.3 The isolated integrity premise

Integrity preservation has been thematized in the literature on transaction processing since its beginnings. We cite (Eswaran et al, 1976): "*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 *the isolated integrity premise*. (Recall that the execution of a transaction T is *isolated* when the state transition effected by T is as if having been executed alone, without concurrent transactions.) From that premise, the well-known result is inferred that then, also all sequentializable schedules of concurrent transactions preserve 'consistency', i.e., integrity. However, similar to the total integrity premise, the isolated integrity premise seems to be illusionary, particularly 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 transactions preserve integrity in isolation. Yet, 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 section 4.

3 EXTENDING REPLICATION WITH INTEGRITY

In 3.1, we identify the replication-integrity conflict for the classes of *Certification-Based Replication* (CBR) and *Weak Voting Replication* (WVR) protocols. In 3.2, we propose an extension to relieve that conflict.

3.1 Replication protocols

Replication protocols use a total order broadcast mechanism (Chockler et al, 2001) for update propagation and replica coordination. According to (Wiesmann et al, 2005), CBR and WVR are the most interesting classes of such protocols. Other classes in (Wiesmann et al, 2005) are unproblematic for integrity management, since they require no coordination between replica. Thus, integrity checking can fully rely on the support of the local DBMS, just like in a non-replicated architecture. Unfortunately, CBR and WVR give rise to several problems with regard to integrity constraints. To analyze these, let us take a closer look at the protocol procedures. Both CBR and WVR process each transaction *T* as follows.

Each transaction *T* is executed by a single *delegate* replica. When *T* requests its commit in the delegate, the writeset of *T* (with CBR also its readset) is broadcast to all replicas. On remote delivery, *T* is checked for concurrency conflicts. If there is a conflict, *T* is aborted. Else, all replicas record *T* as a to-be-committed transaction, for checking conflicts with incoming transactions, and the client is notified of *T*'s success. In WVR protocols (but not for CBR), the delegate then broadcasts a *commit* message for committing *T* in all replicas. Then, *T* attempts to commit in each replica by applying its writeset. If that is impeded (e.g., due to a yet unresolved deadlock in which *T* is involved), then writeset application is re-attempted until it succeeds.

Each protocol is supposed to commit transactions only if they are free of concurrency conflicts. However, deferred integrity checking, done after the protocol has ended, may lead to constraint violation and abortion behind schedule. Worse, a transaction *T* that has been prematurely assumed to be committable may cause other, valid transactions whose data were delivered after *T* to abort due to *T*'s assumed commit. To avoid that, WVR and CBR protocols need to be modified. For WVR, we discuss such a modification in 3.2. For CBR, similar modifications are studied in (Muñoz et al, 2008).

3.2 Extending WVR protocols

The schematic WVR protocol in Figure 1 may be implemented either as a middleware component or as a direct extension of the DBMS core. Anyway, we assume that the DBMS is able to support the isolation level for which the replication protocol has been conceived. Thus, the replication protocol may focus on its native purpose of ensuring replica consistency, and leave local concurrency control to the DBMS.

In Figure 1, let T be a transaction, R the set of alive replicas, re a replica, rd the delegate replica where the protocol is executed, cl the client process, *DB* the local DBMS interface accessed by the protocol, *wset*(T) the writeset of T, and *rset*(T) the readset of T.

A WVR protocol consists of the steps in the pseudo-code of Figure 1. It is structured by three event-driven blocks. In block I (lines 1-3), T is executed locally; its writeset is broadcast upon the event that T is requested to be committed. Blocks II (lines 4-9) and III (lines 10-13) describe what is executed in all replicas. In the former, action is taken upon arrival of the writeset,and T's status is broadcast after validation. In the latter, action is taken upon arrival of the status message, resulting in commit or abort.

The *validate*(.) function checks for read-write or write-write conflicts between T and any other local transaction that has not yet reached its corresponding step 6. Such conflicting local transactions are aborted. Conversely, *validate*(T) returns *abort* if T conflicts with any transaction that has not yet requested commit but has already passed step 6. Note that the *DB.commit*(.) operation results are ignored in the WVR protocol, since it is blindly assumed to be always successful.

```
1:   Execute T.
2:   On T commit request:
3:       TO-brcast(R, <wset(T), rd>)
4:   Upon <wset(T), re> reception:
5:       if (re = rd) then
6:           status_T := validate(T)
             .
             .
             .
7:           R-brcast(R, status_T)
8:           send(cl, status_T)
9:       DB.apply(wset(T))
10:  Upon status_T reception:
             .
11:      if (status_T = commit) then
12:          DB.commit(T)
13:      else DB.abort(T)
```

```
1:   Execute T.
2:   On T commit request:
3:       TO-brcast(R, <wset(T), rd>)
4:   Upon <wset(T), re> reception:
5:       if (re = rd) then
6:           status_T := validate(T)
6a:          if status_T = commit then
6b:              status_T := DB.commit(T)
6c:          else DB.abort(T)
7:           R-brcast(R, status_T)
8:           send(cl, status_T)
9:       else DB.apply(wset(T))
10:  Upon status_T reception:
10a:     if (re ≠ rd) then
11:          if (status_T = commit) then
12:              DB.commit(T)
13:          else DB.abort(T)
```

Figure 1: Weak Voting Protocol, Standard and Extended Version

Deferred constraints checking, which takes place after steps 12 or 13, is done by the DBMS. The DBMS is assumed to issue an exception event and possibly an error report in case of integrity violation. Such exceptions and error messages then will cause the undoing of already committed transactions, which drastically breaches the durability requirement of the ACID principles for transaction handling.

On the right hand side, a simple but effective extension of WVR protocols is displayed. It intercepts and soundly handles exception events due to integrity violations. The extension deals with the possibility that *DB.commit*(.) requests can fail due to integrity violation, upon which an *abort* value is returned. Else, a *commit* value is returned, corresponding to a successful commit. Also notice that the intercepted exceptions do not reach the user-level application, unless the replication protocol decides so.

Lines 6a-6c check the protocol's validation result. If successful, the result is assigned to status_T, yielding an immediate commit of T in the delegate replica. In 6b, T's commit is attempted if no integrity violation has been detected; otherwise, T is aborted. For other abort values corresponding, e.g., to deadlocks or timeouts, the transaction will be reattempted by applying its writeset repeatedly. Thus, commitment of T is attempted before reliably broadcasting the protocol decision. Hence, the delegate replica will know about any constraint violation before commitment, and the replication protocol can react correctly by aborting the transaction in all replicas. Clearly, this coordination of decisions taken by the protocol and the underlying DBMS avoids any replication-integrity conflict. Line 10a lets the delegate replica skip the reception of the message that communicates the termination decision for T, since the delegate replica already knows that decision.

Integrity maintenance comes at a price. Although the extensions in Figure 1 may seem to be minor, they do have a measurable impact in transaction response time. In Figure 1, control was returned to the client before the actual commit was requested. Thus, the transaction completion time was reduced. When integrity is checked, such an optimization is not possible. For an experimental study with comparative measurements and possible optimizations, see (Muñoz et al, 2008).

A potential recovery problem was ignored in (Muñoz et al, 2008): If the delegate replica of a to-be-committed transaction T breaks down between steps 6b and 7, rd had committed the transaction while all others are still waiting for its *commit*/*abort* message. Thus, some other replicas becomes the new delegate, for broadcasting the *commit* message. If all alive replicas then have committed T in a given group view

(Chockler et al, 2001), but the broken one has committed it in a previous group view, recovery is complicated. A typical recovery protocol would re-send T's updates as part of the recovery information. But rd already had executed such updates locally, i.e., they should not be applied again. If rd already breaks in step 6 or 6a, then the updates of T must be included in the recovery tasks. That can be achieved by always transferring the updates and having each recovering replica check whether they had already been committed.

4 INCONSISTENCY TOLERANCE

For enterprise computing, the purpose of business rules is to state and enforce semantic integrity properties of business data and processes. 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, efforts may focus on reducing such inconsistencies. That is the philosophy behind inconsistency-tolerant integrity checking, as revisited in 4.1. In 4.2, we outline a generalization of the results in 4.1 to concurrent transactions in replicated databases.

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. Thus includes the two most common forms of representing integrity constraints: as denials or in prenex normal form. Each method M can be conceived as mappings which takes triples (D, IC, T) as input and outputs either *OK*, which sanctions T as integrity-preserving, or *KO*, which indicates that executing T would violate some constraint. Further, let D^Tdenote the database state obtained by applying the writeset of T to D.

4.1 Waiving the total integrity premise

In (Decker et al, 2006), it is shown that it is possible to waive to total integrity premise for most approaches to integrity checking without any trade-off. Methods which continue to function well when this premise is waived are called *inconsistency-tolerant*. The following example illustrates the idea.

Example 3
Let I and I' be as in example 2. For checking if inserting *emp*(*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, apart from (p, e), also the tuple (p, f) is a row in *proj*. This may at first seem irritating, since I 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, each transaction that leaves all data that were consistent in the before-state consistent in the after-state should not be rejected. And that is precisely what M's output indicates: no instance of any constraint that is satisfied in the before-state is violated by the transaction in the after-state. □

The concept of inconsistency-tolerant integrity checking in (Decker et al, 2006) is formalized as follows.

Definition (*inconsistency tolerance*)
a) A variable x is called a *global variable* in I if x is $\forall$-quantified in I and $\exists$ does not occur left of x.
b) A constraint I' obtained from I by a substitution of global variables in I is called a *case* of I.
c) Let *satcas*(D, IC) denote 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 $C \in$ *satcas*(D, IC), (*) holds:
$M(D, IC, T) = OK \Leftrightarrow C$ is satisfied in D^U (*)

In example 3, 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) $\rightarrow$ *proj*(p, z) of I' is checked by most methods. All irrelevant cases, e.g., the violated case $\leftarrow$ *proj*(p, e), 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 *M* to be correct, (*) is required for each *I* in *IC*, not just for the cases *C* that are satisfied in the before-state. Thus, *M* does not worry about extant constraint violations.

In other words, a method *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 *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 *M* outputs *KO* for any transaction the execution of which would violate a hitherto satisfied instance of some constraint. It is then up to the agent who has called *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 (Decker et al, 2006) 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.

4.2 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 D^T if *I* is satisfied in *D*.

For applying inconsistency-tolerant integrity checking not only to transactions executed in isolation, but also to concurrent transaction in centralized, distributed or replicated systems, let us first restrict the isolated integrity premise to a single but arbitrary integrity constraint *I*. That then guarantees that, for each state *D* such that *I* is satisfied in *D*, and for each transaction *T* that preserves *I* in isolation, *I* will remain satisfied in D^T 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* such that *C* is satisfied in *D*. Now, recall that the condition (*) of the definition of inconsistency tolerance in 4.1 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 in replicated systems with concurrent transactions. More precisely: 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 D^T 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 centralized databases 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 before-state will remain satisfied in the after-state, also for concurrent transactions in replicated databases.

The essential difference of this relaxation to the traditional result is the following. In the relaxed result, isolated integrity preservation only is asked to hold for individual cases. Since integrity checking focuses on cases that are relevant for the writeset of a given transaction *T*, only these cases are guaranteed to remain satisfied. 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 responsible transactions are checked too. If not, such violations are tolerated by an inconsistency-tolerant method which checks *T*.

As nice as our relaxation of the isolated integrity premise may be, it still asks for an unrestricted isolation level with regard to individual cases. Thus, we cannot expect this result to hold if the isolation level is lowered. Since WVR does not compromise the ACID isolation level guarantees, there are no problems for such protocols. However, for SI-based CBR protocols, more research is necessary in order to clarify which consistency guarantees can be given when inconsistency-tolerant integrity checking methods are used.

5 RELATED WORK

The literature on integrity checking in replicated database systems is very scant; solitary exceptions are few and peripheral, e.g., (Su et al, 1987; Veiga, 2003; Okun et al, 2004; Adiba et al, 2007). None of these deals with the problem of integrating the enforcement of business rules with replication protocols, except (Adiba et al, 2007). That approach, however, lacks generality. It is confined to a limited class of business rules and a lazy form of replication that is not adequate for online transaction processing. None of the cited work questions any of the unrealistic premises that we have identified.

For concurrent transactions, the onus of business rules maintenance has, up to now, been on application designers and users. However, that should give way to declarative specifications of automatically supported integrity constraints, just the way they are supported already in centralized, non-distributed database systems. Our approach attempts to be as declarative as possible. That is, business rules should be stated as declarative integrity constraints in SQL, so that everything else can be delegated to the integrity checking module of the DBMS. That module may be built-in or run on top of the DBMS core. In any case, the enforcement of business rules should be as transparent to the user as concurrency, distribution and replication. As opposed to that, established authors of concurrency theory require the near-impossible: that *all* transaction must be programmed so as to guarantee integrity preservation in isolation (Gray, 1981). All related work we have found in the literature adopt this stance unquestioned.

In (Oracle, 2006), a proprietary solution is proposed: ``*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, the semantics of IMMEDIATE is not well-defined. For instance, its meaning seems to be entirely speculative if there are access locks.

A non-proprietary automatism to re-program concurrent transactions such that unwanted conflicts at commit time are avoided is proposed in (Martinenghi et al, 2006). As outlined above, we advocate a different solution: the DBMS should determine autonomously (either by using a built-in procedure or some external device) whether the transition from the before- to the after-state of each given transaction 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.

CONCLUSION

Our goal is to make business rules, expressed as integrity constraints, feasible in replicated databases. We have identified three obstacles that have hitherto prevented to reach that goal: the replication-integrity conflict and the academical escapism to impose the premises of total and isolated integrity.

The replication-integrity conflict can be avoided by suitable modifications of protocols for supporting the replication of databases. Rather than discussing particular solutions for individual protocols, we have proposed a schematic way to modify the popular class of weak voting protocols. Similar modifications for the class of certification-based protocols are described in (Muñoz et al, 2008). Despite their simplicity, these modifications have turned out to be very effective in experimental tests (Muñoz et al, 2008).

For overcoming the traditional belief that integrity can be checked efficiently for given transactions only if the *before*-state is totally satisfied, we have revisited the work in (Decker et al, 2006). 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 inconsistency-tolerant enforcement of business rules for concurrent sequentializable transactions guarantees that no transaction can violate any instance of any constraint that is satisfied in the before-state, if all transactions preserve the integrity of the same instance in isolation. Thus, if any violation happens, then no transaction that has been successfully checked for integrity preservation by an inconsistency-tolerant method can be held responsible for that. Perhaps the most interesting aspect of this result is that it even holds in the presence of extant inconsistencies.

More research is needed for systems in which the isolation level of concurrent transactions is compromised. In particular, it should be interesting to make precise which exactly are the *before*- and *after*-states of concurrent, possibly non-sequentializable transactions, and which consistency guarantees can be made.

ACKNOWLEDGEMENT

This work is partially supported by FEDER and the Spanish MEC grant TIN2006-14738-C02-01.

REFERENCES

Adiba, N., Cochrane, R., Hamel, E., Kulkarni, S., Lindsay, B., 2007. Techniques to preserve data constraints and referential integrity in asynchronous transactional replication of relational tables. *United States Patent 7240054*.

Armendáriz, E., Decker, H., Muñoz, F., González de Mendívil, J.R., 2007. A Closer Look at Database Replication Middleware Architectures for Enterprise Applications. *Proc. TEAA'06,* 69-83. LNCS vol. 4473, Springer.

Chockler, G., Keidar, I., Vitenberg, R., 2001. Group communication specifications: a comprehensive study. *ACM Comput. Surv.* 33(4):427-469.

Christiansen, H., Martinenghi, D., 2006. On simplification of database integrity constraints. *Fund. Inform.* 71(4):371-417.

Eswaran, K., Gray, J., Lorie, R., Traiger, I., 1976. The Notions of Consistency and Predicate Locks in a Database System. *CACM* 19(11):624-633.

Date, C., 2000. What, Not How: The Business Rules Approach to Application Development. *Addison-Wesley*.

Doorn, J., Rivero, L. (eds), 2002. *Database Integrity: Challenges and Solutions.* Idea Group.

Decker, H., Martinenghi, D., 2006. A relaxed approach to integrity and inconsistency in databases. *Proc. 13th LPAR*, 287-301. LNCS vol. 4246, Springer.

Gray, J., 1981. The Transaction Concept: Virtues and Limitations. *Proc. 7th VLDB*, 144-154.

Gray, J., Helland, P., O'Neil, P.E., Shasha, D., 1996. The dangers of replication and a solution. *Proc. SIGMOD'96*, 173-182, ACM Press.

Martinenghi, D., Christiansen, H., 2006. Transaction Management with Integrity Checking. *Proc. 16th DEXA*, 606-615. LNCS vol. 3588, Springer.

Muñoz, F., Ruiz, I., Decker, H., Armendáriz, E., González de Mendívil, R., 2008. Extending Middleware Protocols for Database Replication with Integrity Support. In *Proc. 10th DOA*, 607-624. LNCS vol. 5331, Springer.

Okun, M., Barak, A., 2004. Atomic writes for data integrity and consistency in shared storage devices for clusters. *Future Generation Comp. Syst.* 20(4):539-547.

Oracle, 2006. http://download.oracle.com/docs/cd/B19306_01/server.102/b14220/data_int.htm

Patiño, M., Jiménez, R., Kemme, B., Alonso:, G., 2005. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.* 23(4):375-423.

Ramakrishnan, R., Gehrke, J., 2003. *Database Management Systems*, 3rd edition. McGraw-Hill.

Ross, R., 2003. *Principles of the Business Rule Approach*. Addison Wesley.

Su, S.Y.W., Kumar, A., 1987. Modeling Partitioned and Replicated Databases using an Extended Concept of Generalization. Proc. 11th COMPSAC.

Veiga, L., Ferreira, P., 2003. RepWeb: Replicated web with referential integrity. *Proc. SAC'03*, 1206-1211. ACM Press.

Wiesmann, M., Schiper, A., 2005. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowledge and Data Engineering* 17(4):551-566.