Inconsistency-tolerant Business Rules in Distributed Information Systems

Hendrik Decker [*] and Francesc D. Muñoz-Escoí [*]

Instituto Tecnológico de Informática, UPV, 46022 Valencia, Spain

Abstract. Business rules enhance the integrity of information systems. However, their maintenance does not scale up easily to distributed systems with concurrent transactions. To a large extent, that is due to two problematic exigencies: the postulates of *total* and *isolated* business rule satisfaction. For overcoming these problems, we outline a measure-based inconsistency-tolerant approach to business rules maintenance.

1 Introduction

Business rules, a.k.a. integrity constraints, are an approved means to support the consistency of the modeling [26], the design [21], the development [4], the operation [18], the quality maintenance [22] and the evolution [9] of information systems, in particular if these activities are encased in distributed environments.

Although business rules are meant to avoid inconsistency, they also must work well in the presence of data that violate some of the rules. This requirement of inconsistency tolerance imposes itself even stronger in distributed systems, since the risk of inconsistency is higher there than in centralized systems. The problem addressed in this paper is to enable inconsistency-tolerant business rules not only for individual updates of single-user databases, but also for concurrent database transactions. Concurrency is the norm for distributed systems.

Many methods for checking declarative business rules have been proposed in the literature [17]. Concurrent transactions also have been broadly covered [12] [3] [28]. This paper deals with two common postulates that impede a combination of controlling data consistency by checking business rules, on one hand, and guaranteeing consistency preservation by concurrent transactions, on the other. The first postulate 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 that *the total integrity postulate*. The second is that, for guaranteeing integrity preservation by serializable concurrent transactions, each transaction is supposed to preserve integrity when executed in isolation. We call this requirement *the isolated integrity postulate*. It will turn out that both postulates are unrealistic and indeed not necessary to their full extent in practice.

We point out that the isolated integrity postulate must not be confused with the well-known requirement of an isolated execution of transactions [3], i.e., that

[*] supported by ERDF/FEDER and MEC grants TIN2009-14460-C03, TIN2010-17139, TIN2012-37719-C03-01

concurrent transactions should not step into each other's sphere of control [5], so that anomalies such as phantom updates, dirty or non-repeatable reads are avoided. That requirement usually is complied with by ensuring the serializability of schedules, or some relaxation thereof [28, 27]. However, serializability is independent of the isolated integrity postulate, requiring that integrity be preserved in isolation: while serializability can be guaranteed automatically by the scheduler of the DBMS, the isolated integrity postulate is usually expected to be complied with by the designers, programmers and users of transactions.

The dispensability of the total integrity postulate has been unveiled in [10]. The isolated integrity postulate has been discussed and relaxed in [11]. More precisely, total integrity has been shown to be a superfluous, and the isolated integrity postulate has been significantly relaxed, both by a concept of inconsistency-tolerant integrity checking. That concept was based on 'cases', i.e., instances of integrity constraints. Their violation can be tolerated as long as integrity checking can guarantee that the amount of violated cases does not grow. In [7], a significant generalization has been presented, based on inconsistency measures. The latter permit the tolerance of integrity violation as long as integrity checking can guarantee that the amount of measured inconsistency is not increased by updates. In this paper, we show that measure-based integrity checking also enables a significant weakening of the isolated integrity postulate.

In section 2, we characterize the postulates of total and isolated integrity. In section 3, we recapitulate measure-based integrity checking. We are going to see that it serves to get rid of the total integrity postulate as well as to relax the isolated integrity postulate. In section 4, we address related work, with an emphasis on integrity checking for concurrent transactions. If not specified otherwise, we use conventional terminology and notations for logic databases [1], as well as some basic notions of transaction concurrency [3].

2 Two Unnecessary Postulates

The total integrity postulate is going to be explained in 2.1, the isolated integrity postulate in 2.2. Both postulates are unnecessary and actually bad, since they invalidate predictions of the traditional theories of transaction processing [3] and integrity maintenance [17] if databases contain inconsistent data.

2.1 The Total Integrity Postulate

Integrity checking can be exceedingly costly, unless some simplification method is used [20]. That can be illustrated as follows. (As usual, lower-case letters x, y, z denote variables, in the example below.)

Example 1. Let *emp* be a relation about employees, whose first column is a unique name and the second a project assigned to the employee. The formula $I = \leftarrow \mathit{proj}(x, y), \mathit{proj}(x, z), y \neq z$ is a primary key constraint on the first column of *proj*, a relation about projects, with unique identifiers in the first column. The foreign key constraint $I' = \forall x, y \, \exists z \, (emp(x, y) \rightarrow proj(y, z))$ on the y of *emp* references the primary key of *proj*. Now, assume a transaction T that inserts $emp(Jack, p)$. Most integrity checking methods $\mathcal{M}$ ignore I for checking T, since I does not constrain *emp*. They only evaluate $\exists z \, (emp(Jack, p) \rightarrow proj(p, z))$, or rather its simplification $\exists z \; proj(p, z)$, since $emp(Jack, p)$ becomes *true* by the transaction. If, e.g., (p, e) is a row in *proj*, $\mathcal{M}$ accepts the insertion. If no tuple in *proj* matches (p, z), then $\mathcal{M}$ signals a violation of integrity.

Proofs of the correctness of methods for simplified constraints checking in the literature all rely on the total integrity postulate, 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 postulate is complied with. In particular for applications such as business intelligence, distributed and replicated databases, legacy data maintenance, data warehousing, data federation, etc, a certain amount of inconsistent data that violate constraints in committed states has to be lived with, at least temporarily.

Suppose that, for instance, the constraint I' in Example 1 is violated due to the element $emp(Jack, Illustra)$ in D, after a deletion of the *Illustra* project. Thus, by definition, no traditional method that imposes the total integrity postulate may check T, since not all constraints are satisfied. However, if the project that *Jack* is assigned to is stored in the *proj* relation, T is rightfully sanctioned by all common implementations of integrity checking, as already indicated in Example 1. Example 2, in 3.1, will illustrate essentially the same point.

Hence, the total integrity postulate, which conventionally has always been imposed, does not approve the correctness of integrity checking in practice, since the latter often is performed in the presence of consistency violations. Fortunately, however, that postulate can be abolished without incurring any cost and without losing its essential guarantees, as shown in 3.1.

2.2 The Isolated Integrity Postulate

Integrity preservation has been a pronounced concern already in the early literature on transaction processing. We cite from [12]: "*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 postulate. (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.) Thus, the isolated integrity postulate effectively presupposes the total integrity postulate. From the isolated integrity premise, most authors in the field infer that then, also all sequentializable schedules of concurrent transactions preserve 'consistency', i.e., integrity.

In general, not only the requirement of total integrity, but also the isolated integrity postulate seems to be illusionary, particularly for distributed multi-user databases, let alone for transactions in the cloud. In fact, it is hard to believe than any client who authors 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 preservation of their transactions, although there is no theory to justify their optimism, in the presence of inconsistency. Such a justification is given in Section 3.

3 Inconsistency Tolerance

Business rules state and enforce the integrity of business data. However, inconsistencies due to violations of business rules are unavoidable in practice. Hence, rather than insisting that all business rules must be totally satisfied at all times, it is necessary to tolerate constraint violations. Attempts to reduce or repair such manifestations of inconsistency often are not affordable at update time. Thus, updates should be checkable for consistency preservation even if some constraints are violated. That idea is revisited in 3.1. In 3.2–3.4, 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, $\mathcal{M}$ stand for a database, an integrity constraint, a set of integrity constraints, a transaction and, resp., an integrity checking method. By $D(IC) = \mathit{true}$ and $D(IC) = \mathit{false}$, we denote that IC is satisfied or, resp., violated in D. 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. Moreover, let D^T denote the database state obtained by applying the write set of T to D.

In general, each method $\mathcal{M}$ is a mapping which takes triples (D, IC, T) as input and returns either OK, which means that $\mathcal{M}$ sanctions T as integrity-preserving, or KO, which indicates that T would violate some constraint.

3.1 Getting Rid of Total Integrity

In [10], we have shown that, contrary to common belief, it is possible to get rid of the total integrity postulate for most approaches to integrity checking without any trade-off. Methods which continue to function well when this postulate is renounced are called inconsistency-tolerant. The basic idea is illustrated below.

Example 2. Let I and I' be as in Example 1. Most integrity checking methods $\mathcal{M}$ accept the update *insert* (Jack, p) 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*. That may be somewhat irritating, since the case $\leftarrow \mathrm{proj}(p, e),\ \mathrm{proj}(p, f),\ e \neq f$ of I then is violated. However, this violation has not been caused by the insertion just checked. It has been there before, and the assignment of *Jack* to p should not be rejected just because the data about p are not consistent. After all, it may be part of *Jack*'s new job to cleanse potentially inconsistent project data. In general, a transaction T that preserves the integrity of all consistent data without increasing the amount of extant inconsistency should not be rejected. And that is exactly what $\mathcal{M}$'s output indicates: no case of any constraint that is satisfied in the state before T is committed is violated after T has been committed.

3.2 Inconsistency Measures

Example 2 conveys that each update which does not increase inconsistency can and should be accepted. The following definitions serves to make precise what it means to have an increase or decrease of inconsistency.

Definition 1. $(\mu, \preceq)$ is called an *inconsistency measure* (in short, *measure*) if μ maps tuples (D, IC) to a metric space that is partially ordered by $\preceq$. We may identify a measure $(\mu, \preceq)$ with μ if $\preceq$ is understood.

Example 3. A simple border-case measure β is given by $\beta(D, IC) = D(IC)$, with the ordering $\mathit{true} \prec \mathit{false}$, i.e., constraint satisfaction ($D(IC) = \mathit{true}$) means lower inconsistency than constraint violation ($D(IC) = \mathit{false}$). In fact, β is used by all conventional integrity checking methods, for deciding whether a given transaction T on a database D that satisfies its constraints IC should be accepted (if $D^T(IC) = \mathit{true}$) or rejected (if $D^T(IC) = \mathit{false}$).

A less trivial inconsistency measure, for example, as defined in [8], is the function that maps (D, IC) to the cardinality of the set of cases of violated constraints. Inconsistency can also be measured by taking such sets themselves, as elements of the powerset of all cases of IC, together with the subset ordering.

3.3 Generalizing Inconsistency-tolerant Integrity Checking

Inconsistency-tolerant integrity checking can now be defined as follows.

Definition 2. *(measure-based inconsistency tolerance)*
Let $\mathcal{M}$ be a mapping from triples (D, IC, T) to $\{OK, KO\}$, so that T is either accepted or, resp. rejected, and $(\mu, \preceq)$ an inconsistency measure. $\mathcal{M}$ is called a *sound*, resp., *complete* method for integrity checking if, for each triple (D, IC, T), (1) or, resp., (2) holds.

$$\mathcal{M}(D, IC, T) = OK \;\Rightarrow\; \mu(D^T, IC) \preceq \mu(D, IC). \tag{1}$$

$$\mu(D^T, IC) \preceq \mu(D, IC) \;\Rightarrow\; \mathcal{M}(D, IC, T) = OK. \tag{2}$$

If (1) holds, then $\mathcal{M}$ is also called *measure-based*, and, in particular, *μ-based.*

Definition 2 generalizes the traditional definition of integrity checking significantly, in two ways. Firstly, the traditional measure used for sizing constraint violations is binary, and thus very coarse: IC is either *violated* or *satisfied* in D, i.e., there is no distinction with regard to different amounts of (in)consistency. As opposed to that, the range of an inconsistency measure μ may be arbitrarily fine-grained. Secondly, the total integrity postulate is imposed traditionally, i.e., $D(IC) = \mathit{true}$ is required. As opposed to that, this postulate is absent in Definition 2, i.e., $\mathcal{M}$ does not need to worry about extant constraint violations.

Definition 2 formalizes that a method $\mathcal{M}$ is inconsistency-tolerant if its output OK for a transaction T guarantees that the amount of inconsistency in

(D, IC) as measured by μ is not increased by executing T on D. Moreover, each T that, on purpose or by chance, repairs some inconsistency without introducing any new violation will be OK-ed too by $\mathcal{M}$. Thus, inconsistency-tolerant integrity checking will decrease the amount of integrity violations over time.

Note that it follows by the definition above that each inconsistency-tolerant $\mathcal{M}$ returns KO for any transaction the commitment of which would violate a hitherto satisfied case 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 defensive reaction is to simply cancel and reject the transaction. A more offensive reaction could be to modify ('repair') the database, the constraints or the transaction, so that an increase of the amount of integrity violations is undone. Such measure-based database repairs are dealt with in [6].

3.4 Relaxing Isolated Integrity

To say, as the isolated integrity postulate does, that a transaction T "preserves integrity in isolation", means: For a given set IC of integrity constraints and each state D, each $I \in IC$ is satisfied in D^T if I is satisfied in D.

Now, let us apply the concept of inconsistency-tolerant business rules checking in 3.1 not only to transactions executed in isolation, but also to concurrent transactions. Thus, we abandon the premise "if I is satisfied in D" and weaken the consequence "each $I \in IC$ is satisfied in D^T", as in Definition 2.

In [11], we could show that this is possible for integrity checking methods that preserve all satisfied cases of integrity constraints, while tolerating those that are violated in the state before a given transaction is executed. By an analogous abstraction, the isolated integrity postulate can be weakened as follows.

For each tuple (D, IC), each measure $(\mu, \preceq)$ and each transaction T,

$$\mu(D^T, IC) \preceq \mu(D, IC) \qquad (^*)$$

must hold whenever T is executed in isolation.

Clearly, (*) relaxes the traditional isolated integrity postulate, since neither D nor D^T are required to satisfy all business rules in IC. Rather, T only is required to not increase the measured amount of inconsistency. Thus, by analogy to the proof of Theorem 3 in [11], we arrive at the following generalization.

Theorem 1. Let H be the execution of a serializable history of transactions, T be a transaction in H such that (*) holds whenever T is executed in isolation, D_i be the input state of T in H and D^o be the output state of T in H. Then, $\mu(D_o^T, IC) \preceq \mu(D^i, IC)$ holds.

We point out that this result does not endorse that the inconsistency of all of D and D^T must be measured. On the contrary: measure-based inconsistency-tolerant integrity checking can proceed as for distributed systems without concurrency. In other words, T is committed only if it does not increase the amount of inconsistency, which usually is checked incrementally, without actually assessing the total amount of inconsistency in any state.

Note that the relaxation of the isolated integrity postulate outlined above still asks for the serializability, i.e., a highly demanding isolation level, of all concurrent transactions. Thus, we cannot expect that integrity guarantees of the form (*) would continue to hold in general if the isolation level is lowered. (For a general critique of lowering isolation levels, see [2].) Future work of ours is intended to investigate possible relaxations of the isolation level of concurrent transactions such that sufficient integrity guarantees can still be given.

4 Related Work

Most papers about integrity maintenance do not deal with transaction concurrency. Also the work in [14], which proposes realizations of declarative integrity checking in distributed databases, largely passes by transaction concurrency. On the other hand, most papers that do address concurrency take it for granted that transactions are programmed such that their isolated execution never causes any integrity violation, i.e., they don't care how integrity is ensured.

As an exception, the work documented in [15], addresses both problem areas. However, the proposed solutions are application-specific (flight reservation) and seem to be quite ad-hoc. Also the author of [23] is aware of the problem, and argues convincingly to not be careless about consistency issues. However, with regard to semantic integrity violations in concurrent scenarios, he only exhibits a negative result (the CAP theorem [13]), but does not investigate inconsistency-tolerant solutions. There do exists solutions for reconciling consistency, availability and partition tolerance in distributed systems, e.g., [29] [24]. However, the consistency they are concerned with is either transaction consistency (i.e., the avoidance of dirty reads, unrepeatable reads and phantom updates) or replication consistency (i.e., that all replicas consist of identical copies, so that there are no stale data), not the semantic consistency as expressed by business rules.

A proposal to rewrite concurrent transactions such that conflicts at commit time are avoided is proposed in [16]. The authors outline how to augment transactions with read actions for simplified constraint checking and with locks, so that their serializable execution guarantees integrity preservation. However, ad-hoc transactions are not considered.

For replicated database systems, the interplay of built-in integrity checking, concurrency and replication consistency has been studied in [19]. 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 [19]. 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

Since the beginnings of the field of computational databases, the obligation of maintaining the integrity of business rules in multi-user systems, and thus the avoidance of inconsistency, has rested on the shoulders of designers, implementers, administrators and end users of transaction processing. More precisely, integrity maintenance for concurrent transactions, particularly in distributed systems, is delegated to a multitude of individual human actors who, on one hand, have to trust on each other's unfailing compliance with the integrity requirements, but, on the other hand, usually do not know each other.

The long-term objective toward which this paper has made some steps is that this unreliable distribution of responsibilities for integrity preservation should give way to declarative specifications of integrity constraints that are supported by the DBMS, just like some fairly simple kinds of constraints are supported for sequential transactions in non-distributed database systems.

With this goal in vision, we propose the following. For each transaction T, the DBMS should determine autonomously whether the state transition effected by T preserves integrity or not, and react accordingly. In this paper, we have removed two major obstacles that hitherto may have turned away researchers and developers from striving for such solutions: the postulates of total and isolated integrity preservation.

For overcoming the total integrity postulate, i.e., the traditional misbelief that integrity can be checked efficiently for a transaction T only if the state before T totally satisfies all constraints, we have revisited the work in [10]. There, it has been shown that the total integrity postulate can be waived without further ado, for most (though not all) integrity checking methods.

We have reaffirmed that the advantages of dumping the total integrity postulate even extend to relaxing the isolated integrity postulate. 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 case of any constraint that has been satisfied in the state before committing if all transactions preserve the integrity of the same cases 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 inconsistent data that violate some business rule.

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 locally, states that are committed globally, states that are "seen" by a transaction and states that are "seen" by (human or programmed) agents or clients that have issued the transaction. It should also be able to predict 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. First steps in this direction had been proposed in [11].

Another important, possibly even more difficult area of upcoming research is that of providing inconsistency-tolerant transactions not only in distributed and replicated systems with remote clients and servers, but also for databases in the cloud, for big volumes of data and for No-SQL data stores. These are intended to be the objectives of future projects. So far, some special-purpose solutions exist (e.g., [30]). Their generalizability is doubtful, or at least less than obvious. In fact, the lack of genericity may be a weakness or a strength. After all, a move away from the universality-oriented attitude toward solutions to problems in the field of databases, which was common in the past, seems to be the way of the future, as argued in [25].

References

1. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases.* Addison-Wesley, 1995.
2. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. *Proc. SIGMOD 1995*, 1-10. ACM Press, 1995.
3. Bernstein, P., Hadzilacos, V., Goodman, N.: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
4. Butleris, R., Kapocius, K.: The Business Rules Repository for Information Systems Design. Proc. 6th ADBIS, vol. 2, pp. 64-77. Slovak Univ. of Technology, Bratislava, 2002.
5. Davis, C. T.: Data Processing sphere of control. IBM Systems Journal 17(2):179-198, 1978
6. Decker, H., 2011. Partial Repairs that Tolerante Inconsistency. To appear in *Proc. 15th ADBIS*, LNCS vol. 6909, Springer.
7. H. Decker. Causes of the violation of integrity constraints for supporting the quality of databases. In *Proc. 12th ICCSA, Part V*, volume 6786 of *LNCS*, pages 283–292. Springer, 2011.
8. Decker, H.: New measures for maintaining the quality of databases. In *Proc. 13th ICCSA, Part IV*, volume 7336 of *LNCS*, pages 170–185. Springer, 2012.
9. Decker, H.: Controlling the Consistency of the Evolution of Database Systems. Proc. 24th ICSSEA, Paris, 2012.
10. Decker, H., Martinenghi, D., 2011. Inconsistency-tolerant Integrity Checking. *IEEE Transactions on Knowledge and Data Engineering* 23(2):218-234, 2011.
11. Decker, H., Muñoz-Escoí, F. D., 2010. Revisiting and Improving a Result on Integrity Preservation by Concurrent Transactions. *Proc. OTM Workshops 2010*, 297-306. LNCS vol. 6428, Springer.
12. 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.
13. Gilbert, S., Lynch, N.: Brewer's Conjecture and the feasibility of Consistent, Available, Partition-tolerant Web Services. ACM SIGACT News 33(2):51-59, 2002.

14. H. Ibrahim: Checking Integrity Constraints - How it Differs in Centralized, Distributed and Parallel Databases. *Proc. 17th DEXA Workshops*, 563-568. IEEE 2006.
15. Lynch, N., Blaustein, B., Siegel, M., 1986. Correctness Conditions for Highly Available Replicated Databases. *Proc. 5th PODC*, 11-28. ACM-Press.
16. Martinenghi, D., Christiansen, H., 2006. Transaction Management with Integrity Checking. Proc. 16th DEXA, 606-615. LNCS vol. 3588, Springer.
17. D. Martinenghi, H. Christiansen, H. Decker. Integrity checking and maintenance in relational and deductive databases and beyond. In Z. Ma (ed), *Intelligent Databases: Technologies and Applications*, 238-285. Idea Group, 2006.
18. Morgan, T., 2002. Business Rules and Information Systems - Aligning IT with Business Goals. Addison-Wesley.
19. Muñoz, F., Ruiz, I., Decker, H., Armendáriz, E., González de Mendívil, R., 2008. Extending Middleware Protocols for Database Replication with Integrity Support. In Proc. OTM Conferences, 10th DOA, 607-624. LNCS vol. 5331, Springer.
20. J.-M. Nicolas: Logic for improving integrity checking in relational data bases. *Acta Informatica* 18:227-253, 1982.
21. Novakovic, I., Deletic, V.: Structuring of Business Rules in Information System Design and Architecture. Facta Universitatis Nis, Ser. Elec. Energ. 22(3):305-312, 2009.
22. Pipino, L., Lee, Y., Yang, R.: Data Quality Assessment. CACM 45(4):211-218, 2002.
23. Stonebraker, M.: Errors in Database Systems, Eventual Consistency and the CAP Theorem, 2010. `http://cacm.acm.org/blog/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem`
24. Stonebraker, M.: In search of database consistency CACM vol. 53(10):8-9, 2010.
25. Stonebraker, M.: Technical perspective - One size fits all: an idea whose time has come and gone. *Commun. ACM* 51(12):76, 2008.
26. Taveter, K.: Business Rules' Approach to the Modelling, Design and Implementation of Agent-Oriented Information Systems. Proc. CAiSE workshop AOIS, Heidelberg, 1999.
27. K. Vidyasankar: Serializability. In L. Liu, T. Özu (eds), *Encyclopedia of Database Systems*, 2626-2632. Springer, 2009.
28. Weikum, G., Vossen, G.: *Transactional Information Systems*. Morgan Kaufmann, 2002.
29. Vogels, W.: Eventually Consistent. ACM Queue 6(6)14-19, 2008.
30. Pereira Ziwich, P., Procpio Duarte, E., Pessoa Albini, L.: Distributed Integrity Checking for Systems with Replicated Data. *Proc. ICPADS*, vol. 1, pages 363-369. IEEE CSP, 2005.