Revisiting and Improving a Result on Integrity Preservation by Concurrent Transactions

Hendrik Decker and Francesc D. Muñoz-Escoí *

Instituto Tecnológico de Informática, UPV, Valencia, Spain

Abstract. We revisit a well-known result on the preservation of integrity by concurrent transactions. It says that the serializability of integrity-preserving transactions yields integrity-preserving histories. We improve it in two ways. First, we discuss divergent interpretations and restate them more precisely. Second, we make it applicable in the presence of inconsistency.

1 Introduction

Transactions in distributed information systems are bound to happen concurrently. Each transaction that involves updates may violate the integrity of stored information. We assume that integrity conditions are expressed by declarative constraints in the database schema of the information system.

Concurrent transactions usually are required to be serializable, in order to prevent anomalies that may lead to violations of integrity. The requirement of serializability is based on the well-known result that, if transactions preserve integrity when executed in isolation, then also each serializable concurrent execution of such transactions preserves integrity [8, 2]. Thus, concurrency seems to be harmless, as long as each transaction is ensured to preserve integrity in isolation. However, there are two big problems with this result.

First, there are essentially two different interpretations in the literature, and it is not always clear which is meant. Both are correct, but one of them turns out to be unfair, and both are unfeasible for long histories and ad-hoc transactions.

Second, the result outsources the responsibility of ensuring the integrity preservation of a single transaction T to all preceding and concurrent transactions: If any of them doesn't make sure to preserve integrity on their part, then no guarantees for T are made. Thus, the result is not robust and in fact inapplicable if integrity violations are caused by preceding or concurrent transactions. That intolerance is unrealistic, since most information systems in practice tend to suffer from some (mostly slight) amount of inconsistency, while behaving reasonably well. As it stands, the result is unable to predict such behavior.

In section 2, we formally revisit standard notions and results of concurrency and integrity. In section 3, we propose solutions for the two problems identified above. In section 4, we address related work. In section 5, we conclude. We assume a basic familiarity with concurrency [2] and integrity [18].

* Both authors are supported by ERDF and the Spanish grants TIN2009-14460-C03 and TIN2010-17193

2 Revisiting Concurrency and Integrity

In 2.1 and 2.2, we synthesize notions and formalizations traditionally used in the literature on transaction concurrency and data integrity. In 2.3, we recapitulate the mentioned result about the integrity preservation by concurrent transactions.

2.1 Concurrency

Drawing from [8, 2, 17], we introduce, in 2.1.1 - 2.1.3, our notions of 'resource' (a.k.a. 'data item'), 'fixed-time' and 'dynamic' database state, 'action' (a.k.a. 'operation'), 'transaction', 'history' (a.k.a. 'schedule') and 'serializability'. Our concurrency model does not recur on physical data items nor on fixed-time states, but on truth values of relational tuples in dynamic states of information systems.

2.1.1 Resources, States, Transactions

Throughout, we assume that a well-defined schema of the database that underlies a considered information system is always tacitly given. Also, we assume a universal language $\mathcal{L}$ by which the contents of the relational tables and the integrity constraints of a schema are described.

A *resource* (sometimes also called 'data item') is a unit of storable information accessed by transactions. As in [17], the only resources considered in this paper are the elements of the Herbrand base of $\mathcal{L}$, i.e. all facts expressible in $\mathcal{L}$.

A *state* of an information system is a mapping from the set of all resources to $\{true, false\}$. A state is *partial* if the mapping is partial. For a state S and a resource r, we say that the value of r is *known* in S if it is known to be either *true* or *false* in S, for which we also write $S(r) = true$ or, resp., $S(r) = false$. Otherwise, r is said to be *unknown* in S.

In [2], "the values of the data items at any one time comprise the state". We call such states *fixed-time states* ('dynamic' states are defined in 2.1.2). For convenience, we sometimes identify a fixed-time state with the point of time at which it is fixed. States are changed over time by actions.

An *action* is an atomic operation on precisely one resource, except *begin* and *end*, as defined below: *begin* acts on no resource, *end* may act on many resources.

Each action takes place at precisely one point of *system time*, which we assume to be an unbounded, linear sequence of discrete *time points*.

A *transaction* is an atomically executed, finite set of actions. (We may speak elliptically of a transaction T when in fact we mean an execution of T.) Each transaction T consists of precisely one *begin*, precisely one *end* of the form *commit* or *abort*, and a finite set of *accesses*, each of the form *read*(r) or *write*(r), where r is a resource. The *begin* (*end*) of T is earlier (resp., later) than each access of T. The *commit* of T means that the last write to each resource written by T is confirmed. The *abort* of T means that all writes of T are undone. Two actions are said to be *conflictive* if both access the same resource and at least one of them is a *write*. Conflictive actions in T are never executed at the same time.

Distinguished fixed-time states are the states at the time a transaction T begins and, resp., ends, which we denote by S_T^b and, resp., S_T^e.

To *read* a resource r means to query if r is *true* or *false* of a given state. So, queries correspond to transactions that read the resources needed to return answers. To *write* r means to either *insert* or *delete* r, i.e., effect a state change such that r becomes *true* or, resp., *false*. For actions *read*(r) and *write*(r) of a transaction T, we also say that T *accesses* (*reads* or, resp., *writes*) r.

For a transaction T, let $\mathbb{C}_T$ denote the set of transactions that are concurrent with T, i.e., that execute at least one action in the interval between the begin and the end of T. In particular, $T \in \mathbb{C}_T$.

Transactions that abort are supposed to leave the database unchanged. Thus, they cannot cause a violation of integrity. So, for each transaction, we can assume that it does not abort unless its commitment would violate integrity. Other aborts, due to hardware failure, wrong input, deadlock, etc., are not considered.

2.1.2 Histories, I/O States

Informally, a history is a possibly concurrent execution of transactions, i.e., actions of several transactions may be executed at the same or interleaved points of time. Formally, a *history* H of a set of transactions $\mathbb{T}$ is a partial order of the union of all actions of all $T \in \mathbb{T}$, such that, for each $T \in \mathbb{T}$ and each pair of actions (A, A') in T such that A is before A' in T, A also is before A' in H, and conflicting actions in H are never executed at the same time. Two actions in H *conflict* if they access the same resource r and at least one of them writes r.

We may say 'T *in* H' if H is a history of a set of transactions $\mathbb{T}$ and $T \in \mathbb{T}$. For each T in H, we assume that also each $T' \in \mathbb{C}_T$ is in H, since, otherwise, scheduling may not be able to take all possible conflicts into account. Thus, histories may be arbitrarily long, particularly in systems with 24/7 services that may always be busy. Distinguished fixed-time states at which no access takes place are the states at the time of the earliest *begin* and the latest *end* in H, denoted by S^b_H and, resp., S^e_H. S^e_H is also called the *final state* of H.

For a resource r and a point of time t, the *committed value of* r *at* t *in* H is defined as the value of r as committed most recently by some T in H. Thus, for the commit time t_c of T, $t_c \leq t$ holds, and no transaction in H other than T commits r at any time in the interval $[t_c, t]$.

There are *dynamic* states that are not necessarily fixed-time, e.g., 'states seen by transactions' [8], or 'global states' [9, 4]. Dynamic states consist of values of resources committed at different but related points of time in some history H.

For example, *committed states*, defined by the committed value of each resource at some time t in H, are dynamic. In general, the committed state at time t is different from the fixed-time state at t.

The dynamic states defined next, collectively called *I/O states*, are partial, since transactions usually 'see' (access) only part of the database. In 2.1.3, we use I/O states for characterizing serializability, and in 3.1 for stating precisely which state transition is meant when we discuss the integrity preservation of T.

For a transaction T and a resource r, the value of r in the *input state* S^i_T of T is the committed value of r immediately before T accesses r first. The value

of r in the *output state* S^o_T of T is the value of r immediately after T accessed r last. If a resource is not accessed by T, its values in S^i_T and S^o_T remain unknown.

Clearly, $S^i_T = S^b_T$ and $S^o_T = S^e_T$ if T is executed in isolation. I/O states are dynamic, since they may not exist at any fixed point of time. In particular, S^i_T and S^o_T may be different from S^b_T or, resp., S^e_T.

For instance, a resource may be committed after the begin of T but before T accesses it first. Or, a resource, after having been accessed last by a read operation of T, may be written by some T' in $\mathbb{C}_T$ before T ends. Also, S^i_T and S^o_T are not necessarily identical to any committed state at any point of time.

For example, consider distinct resources r, r' and a history H of transactions *T0*, *T1*, *T2* which begin at a time, then *T0* inserts r and r' at a time and commits, then *T1* reads r, then *T2* deletes r and r' at a time and commits, then *T1* reads r' and commits. Clearly, r is *true* and r' is *false* in $S^i_{T1} = S^o_{T1}$, which is not a committed state at any time of H. Yet, in general, S^i_T and S^o_T are the first and, resp., the last state 'seen by' T.

I/O states facilitate the modeling of long histories, e.g., for 24/7 applications, where the initial or terminal committed states at the time of the begin or the end of histories may be forgotten or out of sight, respectively. Also the modeling of histories with relaxed isolation requirements is easier with I/O states, since they do not necessarily coincide with committed states.

2.1.3 Serializability

The serializability of a history H (usually taken care of transparently by a module called *scheduler*) prevents anomalies (lost updates, dirty reads, unrepeatable reads) that may be caused by concurrent transactions in H [2].

A history H is *serial* if, for each pair of distinct transactions T, T' in H, the begin of T is before or after each action of T', i.e., transactions do not interleave. Intuitively, a serializable history H "has the same effect as some serial execution" of H, where the "effects of a history are the values produced by the Write operations of unaborted transactions", thus preventing that actions of concurrent transactions would "interfere, thereby leading to an inconsistent" state [2]. Anomalies are not the only possible cause of integrity violation. Thus, serializability helps to avoid some, but not all possible integrity violations.

There are several definitions of serializability in the literature [21]. The following one generalizes view serializability [2], but still ensures that, for each serializable history H, the same effects are obtained by some serial execution of H. A history H is called *serializable* if the output state of each transaction in H is the same as in some serial history H' of the transactions in H such that $S^b_{H'} = S^b_H$. For example, the history of *T0*, *T1*, *T2* in 2.1.2 is not serializable.

In practice, less permissive but more easily computable definitions of serializability are used. Locking, time stamping or other transaction management measures may be used for implementing various forms of serializability [8, 2].

2.2 Integrity

In 2.2.1 - 2.2.2, we revisit the notions of 'update', 'integrity constraint' and 'case'. The latter is fundamental for inconsistency-tolerant integrity preservation by concurrent transactions, as addressed in 3.2.2. Inconsistency tolerance guarantees that all constraints that are satisfied before a transaction remain satisfied afterward, even if some constraints are violated before.

2.2.1 Updates, Constraints, Cases

Integrity is endangered whenever an update U to be committed changes a state S to an *updated* state, which we denote by S^U.

An *update* of a state S is a bipartite set (Del, Ins) of resources such that the *deletes* in Del and the *inserts* in Ins are disjoint. S^U is defined by mapping each delete in Del to *false*, each insert in Ins to *true*, and the value of each other resource is as in S. The *writeset* W_T of a transaction T is the set of all writes w in T of the form *delete*(r) or *insert*(r) such that any other write of r in T is earlier than w. Let U_T denote the update corresponding to W_T.

An *integrity constraint* I (in short, *constraint*) is a sentence in L_S which states a condition that is expected to hold in each I/O state. W.l.o.g., we assume that each constraint is represented in *prenex form*, i.e., all quantifiers precede all predicates and connectors. That includes constraints in denial form [14] and prenex normal form [19]. An *integrity theory* is a set of integrity constraints.

A $\forall$-quantified variable in a constraint is called *global* if its quantifier is not preceded by any $\exists$ quantifier. For a substitution ζ of the global variables of a constraint I, a constraint of the form $I\zeta$ is called a *case* of I. Clearly, each constraint subsumes each of its cases, and each constraint is a case of itself.

If a constraint I is violated by some transaction, then often only a single case of I is violated, while all other cases of I remain satisfied. As we shall see in 3.2, cases are very useful for obtaining an inconsistency-tolerant generalization of the results in 3.1 and 3.2. It does not insist on the total integrity satisfaction of all constraints in all committed states, as opposed to traditional results.

2.2.2 Integrity Satisfaction, Violation and Preservation

For an integrity theory IC, a state S *satisfies* IC if each constraint I in IC is satisfied in S. Let $S(I) = sat$ denote that S satisfies I, and $S(I) = vio$ that S violates I. Further, let $S(IC) = sat$ denote that S satisfies IC, and $S(IC) = vio$ that S violates IC. Common synonyms for 'integrity satisfaction' and 'violation' are 'consistency' and, resp., 'inconsistency'. There are several non-equivalent definitions of integrity satisfaction and violation in the literature. A natural one, which we adopt, is to logically evaluate each constraint according to the truth values of resources in S. Thus, $S(I) = sat$ ($S(I) = vio$) if I evaluates to *true* (resp., *false*) in D, and $S(IC) = sat$ if $S(I) = sat$ for each $I \in IC$, else $S(IC) = vio$.

For a constraint I, a transaction T is said to *preserve* I *in isolation* if, for each state S such that $S(I) = sat$, also $S^{U_T}(I) = sat$ holds. For an integrity theory

IC, T is said to *preserve IC in isolation* if T preserves each constraint in IC in isolation. The phrase 'in isolation' can be omitted whenever it is understood. We may also say that T *preserves integrity* when I or IC is understood.

2.3 A Well-known Result

As already indicated, a well-known result of concurrency theory seems to provide an immediate solution to the problem stated in the introduction. We cite this result from [2]: "If each transaction preserves consistency, then every serial execution of transactions preserves consistency. This follows from the fact that each transaction leaves the information system in a consistent state for the next transaction. Since every serializable execution has the same effect as some serial execution, serializable executions preserve consistency too." In other words: if each of several transactions preserves integrity in isolation, then integrity is preserved also when these transactions are executed concurrently in a serializable history. Let us represent this by the following schematic rule:

$$\textit{isolated integrity} + \textit{serializability} \Rightarrow \textit{concurrent integrity} \quad (*)$$

3 Improving the Predictions of (*)

We are going to assess the shortcomings of (*) as mentioned in section 1 and improve its predictions. In 3.1, we observe that the more common of two divergent interpretations of (*) unfairly ignores the committed states of transactions whose commit is not the last one in a given history. In 3.2, we argue that both interpretations of (*) are questionable for long histories and ad-hoc transactions. We then state refinements of (*) that are applicable for long histories, ad-hoc transactions, and extant integrity violations.

3.1 Divergent Interpretations

For a serializable history H the transactions of which preserve integrity in isolation, there are two valid interpretations of the conclusion of (*). One is that integrity is satisfied in S^e_H if it is satisfied in S^b_H. We call that *final-state integrity* (many authors speak of 'final state consistency', e.g., [16]). The other interpretation is that, for each transaction T in H, the transition from S^i_T to S^o_T preserves integrity. A premise of both is that S^b_H satisfies integrity. However, note that this premise may not be applicable in long histories, the beginning or end of which may be unknown or out of sight.

Final-state integrity has been investigated, e.g., in [20, 15, 12]. That interpretation unfairly ignores output states of transactions that commit before the last commit of H. The final-state interpretation makes no integrity guarantees for such states, although they should be trustable for applications that 'see' them.

The second meaning of (*), as identified above, is what we are after in this paper. But, which state transition is meant by asking whether a transaction T

preserves or violates integrity when executed concurrently? It cannot be the transition between S_T^b and S_T^e, since each fixed-time state may be inconsistent, due to possible intermediate integrity violations by concurrent transactions. Rather, T transfers its input state S_T^i to its output state S_T^o. But, can it be said that the transition between partial states S_T^i, S_T^o preserves or violates integrity? Indeed, it can, if the values of all resources to be read for determining if integrity is preserved or violated are known. That, however, we can simply assume, since each of these resources would need to be read by, and thus known to T if T would have to cater by itself for preserving integrity in isolation. Hence, we can re-state both interpretations of (*) as follows.

Theorem 1. Let IC be an integrity theory, H a serializable history such that $S_H^b(IC) = sat$, and T a transaction in H. Further, let each transaction in H preserve IC in isolation. Then, the following holds.

a) $S_H^e(IC) = sat$, i.e., integrity is satisfied in the final state of H.

b) $S_T^o(IC) = sat$, i.e., integrity is satisfied in the output state of T. □

Clearly, theorem 1*a*) corresponds to the first, 1*b*) to the second of the mentioned interpretations of (*). Note that the premises of theorem 1 also entail that each committed state at any time in H satisfies integrity, by the same argument as cited in 2.3 from [2]. Similarly, the premises of theorem 1 entail that also S_T^i satisfies integrity, since, for each resource r accessed by T, $S_T^i(r)$ is the committed value of r at the time it is accessed first by T.

3.2 Robust Integrity Guarantees for Concurrent Transactions

The preconditions of Theorem 1 are quite strong. They demand that IC be totally satisfied from the beginning of the history, and that all transactions make sure that all of IC remains satisfied. However, for obtaining desirable consistency guarantees for T, much less needs to be required.

In 3.2, Theorem 2 shows that it is not necessary, as in Theorem 1, to require integrity guarantees from any transaction that commits before T begins nor from any transaction that is concurrent with T. In 3.2.2, Theorem 3 improves Theorem 2 by abandoning the unnecessarily strict requirement that integrity be satisfied without exception at the beginning of T. Theorem 3 states that all cases of constraints in IC that are satisfied in the input state of T will remain satisfied in its output state if T preserves integrity if it were executed in isolation. Again, note that no such requirement is made for any transaction that precedes T or is concurrent with it. Thus, Theorem 3 is an unprecedented result about inconsistency-tolerant integrity preservation through concurrent transactions.

3.2.1 Independence of Preceding and Concurrent Transactions

Theorem 1 requires that integrity be satisfied in the initial state S_H^b of H. However, for a long history H, S_H^b may be out of reach, i.e., the values and hence

the integrity status of S^b_H may be unknown. Moreover, $1b$) requires that each transaction in H preserves integrity in isolation. That is scary, particularly in multi-user databases where different agents may issue transactions independently of each other. Hence, the guarantees of integrity preservation made for T in $1b$) are betting on something that may be beyond their control. In fact, having to trust on the integrity preservation of preceding or concurrent transactions issued by other, possibly unknown agents is unacceptable.

Hence, is is desirable to relax the related premises of theorem 1, as done in the following result. Its validity is justified by the argument in [2] as cited in 2.3. It also applies to long histories in 24/7 systems. Moreover, it is independent of the preservation of integrity in isolation by transactions other than T in H.

Theorem 2. Let IC be an integrity theory, H a serializable history, and T a transaction in H such that $S^i_T(IC) = sat$ and T preserves IC in isolation. Then, $S^o_T(IC) = sat$. □

The essential differences between theorems $1b$) and 2 are as follows. The premise in theorem 1 that $S^b_H(IC) = sat$ is replaced by the premise in theorem 2 that $S^i_T(IC) = sat$. Thus, the initial state of H, which may be out of reach, does no longer have to be considered. Further, the premise in theorem 1 that each transaction in H preserves IC in isolation is abandoned in theorem 2. It is needed in theorem 1 in order to ensure that integrity remains satisfied from the beginning to the end of H. Theorem 2 predicts that integrity remains satisfied from the input to the output state of an arbitrary transaction T in H, and therefore does not need the abandoned premise. The conclusions are the same, but in theorem $1b$), the conclusion holds for each transaction in H, in theorem 2 just for T. Thus, theorem 2 only considers the state transition effected by T. It does not depend on the integrity satisfaction in states at any time before T begins, nor on the integrity preservation by any other transaction.

3.2.2 Inconsistency-tolerant Integrity Preservation

The premise $S^i_T(IC) = sat$ of theorem 2 effectively requires that each case in IC be satisfied in S^i_T. However, the prediction of a successful integrity preservation by T should not depend on cases that are irrelevant to U_T. Such cases may be violated by transactions that precede or are concurrent with T. More generally, it is desirable to tolerate any extant violations, even of relevant cases (e.g., of 'soft' constraints). The following result relaxes the overly exigent premise $S^i_T(IC) = sat$. Theorem 3 focuses on cases that are satisfied in S^i_T, without requiring that all of them be satisfied, as opposed to theorem 2.

Theorem 3. Let IC be an integrity theory, H a serializable history and T a transaction in H that preserves integrity in isolation. Then, for each case C in IC such that $S^i_T(C) = sat$, it follows that $S^o_T(C) = sat$.

Proof. The set of cases such that $S^i_T(C) = sat$ is an integrity theory that is satisfied in S^i_T. Hence, theorem 3 follows from theorem 2. □

Theorem 3 is an *inconsistency-tolerant* result, in that it makes predictions of integrity preservation by a concurrently executed transaction T, while admitting integrity violations in S_T^i and S_T^o, even of relevant cases. In contrast to theorem 1*b*), already theorem 2 is inconsistency-tolerant, in a sense: as long as $S_T^i(IC) = sat$ holds, any integrity violation in any state of H before S_T^i is immaterial. Theorem 3 goes beyond the inconsistency tolerance of theorem 2 since the former also tolerates any amount of extant violations of constraints in S_T^i.

4 Related work

In early work [11, 7, 13, 8, 1, 10], a distinction is made between integrity violations caused either by anomalies of concurrency or semantic errors. In [7, 13], concurrency is not dealt with any further. In [11, 8, 1, 10], integrity is not looked at in detail. Also in later related work, either concurrency or integrity is largely passed by, except papers that address final-state integrity (cf. 3.1), and [3, 17].

In [17], it is described how to automatically augment concurrent write transactions with additional read actions for simplified integrity checking, and with locks to protect those actions, so that integrity preservation can be guaranteed for serializable executions. However, neither long histories nor ad-hoc transactions are not considered in [17].

The author of [3] 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 [3].

In none of the cited papers, inconsistency tolerance is an issue.

5 Conclusion

We have defined states 'seen by' concurrent transactions as dynamic partial states called I/O states. They typically contain unknown values than may violate integrity. Thus, an inconsistency-tolerant approach is needed. Based on I/O states, we have scrutinized, restated and generalized the classic result that integrity preserved in isolation is sufficient for preserving integrity concurrently.

The classic result is always stated informally, while our versions are more formal and lucid. Our versions also are more practical because they allow for ad-hoc transactions, long histories and the toleration of extant inconsistency.

We have only considered flat transactions. Yet, it should be interesting to study the preservation of integrity when there are read-only subtransactions. That has been already done in [6], but without much attention to concurrency. Alternatively, a concurrent, history-wide or perpetual built-in transaction could be conceived, for checking if user transactions preserve integrity. We intend to study these issues in future work. Also, we intend to take recovery issues into account, which are important for distributed systems but have not been addressed in this paper.

References

1. R. Bayer. Integrity, Concurrency, and Recovery in Databases. *Proc. 1st ECI*, LNCS vol. 44, 79-106. Springer, 1976.
2. P. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
3. S. Böttcher. Improving the Concurrency of Integrity Checks and Write Operations. *Proc. 3rd ICDT*, LNCS vol. 470, 259-273. Springer, 1990.
4. M. Chandy, L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM TOCS* 3(1):63-75, 1985.
5. H. Decker, D. Martinenghi. Inconsistency-tolerant Integrity Checking. *To appear in Transactions of Knowledge and Data Engineering.* Abstract and preprints at *http://www.computer.org/portal/web/csdl/doi/10.1109/TKDE.2010.87.*
6. A. Doucet, S. Gançarski, C. León, M. Rukoz: Checking Integrity Constraints in Multidatabase Systems with Nested Transactions. *Proc. 9th CoopIS*, LNCS vol. 2171, 316-328. Springer, 2001.
7. K. Eswaran, D. Chamberlin. Functional Specification of a Subsystem for Data Base Integrity. *Proc. 1st VLDB*, 48-68. ACM Press, 1975.
8. K. Eswaran, J. Gray, R. Lorie, I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *CACM* 19(11):624-633, 1976.
9. M. Fischer, N. Griffeth, N. Lynch. Global States of a Distributed System. *IEEE Trans. Software Eng.* 8(3):198-202, 1982.
10. G. Gardarin. Integrity, Consistency, Concurrency, Reliability in Distributed Database Management Systems. In C. Delobel, W. Litwin (eds), *Distributed Databases*, 335-351. North-Holland, 1980.
11. J. Gray, R. Lorie, G. Putzolu. Granularity of Locks in a Shared Data Base. *Proc. 1st VLDB*, 428-451. ACM Press, 1975.
12. P. Grefen. Combining Theory and Practice in Integrity Control: A Declarative Approach to the Specification of a Transaction Modification Subsystem. *Proc. 19th VLDB*, 581-591. Morgan Kaufmann, 1993.
13. M. Hammer, D. McLeod. Semantic Integrity in a Relational Data Base System. *Proc. 1st VLDB*, 25-47. ACM Press, 1975.
14. R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.
15. L. Lilien, B. Bhargava. A Scheme for Batch Verification of Integrity Assertions in a Database System. *IEEE Trans. Software Eng.* 10(6):664-680, 1984.
16. B. Lindsay. Jim Gray at IBM – The Transaction Processing Revolution. *Sigmod Record* 37(2):38-40, 2008.
17. D. Martinenghi, H. Christiansen. Transaction Management with Integrity Checking. *Proc. 16th DEXA*, Springer LNCS vol. 3588, 606-615. Springer, 2005.
18. 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.
19. J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica* 18:227-253, 1982.
20. A. Silberschatz, Z. Kedem. Consistency in Hierarchical Database Systems. *JACM* 27(1):72-80, 1980.
21. K. Vidyasankar. Serializability. In L. Liu, T. Özu (eds), *Encyclopedia of Database Systems*, 2626-2632. Springer, 2009.