Data Consistency: Toward a Terminological Clarification

Hendrik Decker[1], Francesc D. Muñoz-Escoí[1], and Sanjay Misra[2](✉)

[1] Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Valencia, Spain
[2] Covenant University, Ota, Nigeria
sanjay.misra@covenantuniversity.edu.ng

Abstract. ‘Consistency’ is an ‘inconsistency’ are ubiquitous term in data engineering. Its relevance to quality is obvious, since ‘consistency’ is a commonplace dimension of data quality. However, connotations are vague or ambiguous. In this paper, we address semantic consistency, transaction consistency, replication consistency, eventual consistency and the new notion of partial consistency in databases. We characterize their distinguishing properties, and also address their differences, interactions and interdependencies. Partial consistency is an entry door to living with inconsistency, which is an ineludible necessity in the age of big data.

1 Introduction

In the field of databases, the meaning of the word ‘consistency’ is overloaded with multiple, often unclear meanings, as revealed by googling or looking it up in Wikipedia [59]. The terminological disarray becomes even more discomforting when trying to clarify the meaning of ‘consistency’ in terms of ‘data quality’. Consistency is unanimously considered one of the most important ‘aspects’ or ‘dimensions’ of data quality [61] [21]. Occasionally, both terms even are identified [15]. In general, however, different people (authors as well as readers) may mean different things with ‘consistency’, without being explicit or clear about the differences.

Another frequently used synonym for ‘consistency’ is ‘correctness’. But, without further explanation, that just replaces one unclear term by another. So, two questions arise that beg for an answer: What precisely is meant by ‘consistency’, and how to avoid, eliminate or at least contain inconsistency. In this paper, we propose some prolegomena for answering the first one. The second is for future work, but first steps for answering it are taken already in Sections 6 and 7.

Necessary conditions for data consistency can be expressed by an *integrity theory*, i.e., a finite set of *integrity constraints* (sometimes also called *assertions*, or, in database normalization theory, *dependencies*). These are sentences in the database language that are supposed to evaluate to *true* (i.e., to be *satisfied*) in each committed state of the database. We are going to clarify the meaning

H. Decker and F.D. Muñoz—supported by the Spanish MINECO grant TIN 2012-37719-C03-01.

of 'consistency' in connection with updates that may affect the truth value of integrity constraints associated to the given database.

Four situations or reasons can be distinguished, why or when updates may lead to consistency violation. One is simply that a bad update directly contradicts what is required by some constraint, e.g., the insertion of *married*(*fred*, *fred*) which goes against the constraint that nobody can marry herself or himself. The second situation which may lead to constraint violations are the well-known 'update anomalies' that are due to an insufficient observance of data redundancies or dependencies that have not been eliminated by schema normalization [5]. The third kind of occurrence of integrity violation is due to equally well-known 'update phenomena' caused by deficient concurrency control [10]. The fourth is due to bad management of data distribution or replication [55].

The mentioned reasons of inconsistency are conductive to the different kinds of consistency studied in the remainder. In particular, we are going to address *semantic consistency*, *transaction consistency*, *distribution consistency*, *replication consistency* and *eventual consistency*. Also, we propose the new notion of *partial consistency*, which alleviates the abrasiveness of inconsistency. We conclude with an outlook on coping with big data inconsistency and more future work.

We are not going to deal with normalization except to emphasize that there is no normal form that could guarantee any of the consistency properties we are going to deal with. In fact, normalization can be very helpful, but is neither sufficient nor necessary for consistency. Moreover, we are not going to deal with physical consistency (a.k.a. disk consistency, or file integrity i.e., the inviolacy of binaries) [52], nor with network consistency [43], nor with the consistency or integrity of data transmission or communication (a topic of coding theory and cryptography) [51], nor with any other integrity issue of concern to database security, such as fraudulent tampering, authenticity or trustworthiness of data [50]. Nor is 'relational consistency' (an issue in constraint programming [16]) of interest in this paper. Also, we do not discuss issues related to the provenance, accuracy or truthfulness of data, concerning the consistency between the (real or imaginary) world that the database is supposed to model and the database content itself. For instance, if the *born* attribute b in a database entry about a person p is not p's actual birth date, then we do not count that mistake as an inconsistency, as long as there is no other stored information which would contradict b. Such issues are studied in fields such as data lineage [38], truth finding [40], truth discovery [62] and data fusion [26].

2 Semantic Consistency

Semantic consistency is a property of database states. Thus, for a given database schema, semantic consistency can be identified with a subset of all possible states. Ideally, the predicate that corresponds to the characteristic function of that subset is described by an integrity theory.

In general, however, semantic consistency is not expressed completely by explicit declarative integrity constraints in the database schema. In fact, the integrity conditions of an application even may not all be cleanly documented. Worse, not all developers of applications may be aware of every semantic constraint. Hence, semantic consistency tends to be hard to guarantee, in general.

Instead of an automated enforcement of declarative constraints, semantic consistency is often realized by nested subtransactions, or firing triggers, or running stored procedures, or executing inline code of application programs, or compensating transactions, the latter in case an update is detected post-factum to have violated integrity. Such procedural ways of integrity enforcement are known to be error-prone and hard to maintain.

In this paper, we identify, for simplicity, semantic consistency with what is expressed (or is expressible) in the integrity theory that is (implicitly or explicitly) associated to the database. Thus, semantic consistency means that all integrity constraints in the given integrity theory are satisfied.

Semantic consistency is also known under the name of *data integrity* (which, however, also suffers from a fuzzy overload of different meanings, in- and outside of the literature on data quality [53] [60]; occasionally, semantic integrity is even identified with data quality [54]).

Ideally, semantic consistency is enforced automatically, by the DBMS or some module on top of its core, so that transaction designers, application programmers and users need not be asked to pay attention to the preservation of integrity. Automatic integrity enforcement usually sanctions an update only if it does not produce any constraint violation. The common built-in enforcement of some specific forms of declarative consistency conditions such as primary and foreign key constraints is an imperfect realization of that ideal. Imperfect because, among others, more general forms of constraints usually are not supported by the systems on the market. For query optimization or mere documentation, some do support a declarative assertion of more complex constraints (e.g., Oracle's RELY construct [46]), but not their enforcement. More comprehensive methods to support an automated checking of declarative integrity constraints, such as described, e.g., in [44] [17], have not yet found their way into marketed products.

3 Transaction Consistency

Simply put, transaction consistency means concurrency-transparent semantic consistency. That is, the database system ensures that each complete history H of concurrent transactions preserves semantic consistency if each transaction T in H preserves semantic consistency whenever T is executed alone [10].

In Section 2, semantic consistency has been characterized as a property of database states. Transaction consistency is a property of state transitions. More precisely, each successful execution of a transaction (which may consist of a single update or comprise a partially ordered set of updates) effects a state change that is supposed to preserve semantic consistency. Thus, for non-concurrent transactions, all of what has been said in Section 2 applies, without further worries.

However, concurrent transactions may violate the semantic consistency in a way that is not to blame on possible integrity violations by individual transactions, but on some harmful interleaving of actions of different transactions. Such interleavings may lead to well-known problems of concurrency such as dirty or non-repeatable reads, lost updates and other anomalies. These are not necessarily problematic by themselves, but only if they lead to the violation of semantic consistency. To avoid such anomalies, *serializability* was invented [34].

Transaction consistency sometimes is wrongly identified with serializability, i.e., the equivalence of the execution of a history H of transactions with a one-by-one execution of all transactions in H. However, neither semantic consistency nor transaction consistency are guaranteed by serializability alone, nor does a history H the transactions of which preserve semantic consistency necessarily entail its serializability [39]. Yet, given a consistent input state, it is plausible that H and in particular each transaction T in H teminate consistently if H is serializable and each *solitary* execution of T (i.e., running T alone) preserves semantic consistency [10]. That can be expressed schematically as follows.

(*) serializability + solitary consistency $\Rightarrow$ transaction consistency

In Section 2, we have pointed out practical problems of warranting semantic consistency. Hence, the solitary consistency preservation of each transaction in a history is an Achilles heel of (*).

In 3.1 – 3.4, we relate transaction consistency to the popular *ACID* properties [35], by discussing them with regard to serializability and semantic consistency.

3.1 ACID

Brewer is quoted saying that the acronym *ACID* is "contrived [...] much more than people realize", that Jim Gray "admitted that *ACID* was a stretch", and "A and D have high overlap and the C is ill-defined" [14]. Nevertheless, many authors identify *ACID* with either serializability or (*) or transaction consistency.

The following brief explanation of *ACID* leans on [10]: A stands for *atomicity*, C for *consistency*, I for *isolation*, and D for *durability*. Atomicity means that each transaction in H terminates either by committing or aborting. Consistency means semantic consistency as characterized in Section 2. Isolation means that the interleaved execution of transactions in H does not harm semantic consistency. Durability means that, once committed, the effects of a transaction will remain persistent, or will be recovered after temporary unavailability, until they are modified, undone or overridden by any subsequent transaction.

Without concurrency, I is trivially satisfied, both A and D can be ensured fairly straightforwardly, and C can be handled as indicated in Section 2. For concurrent transactions, however, A, C, I and D are interrelated much more than without concurrency. For instance, imagine a transaction that infringes atomicity by terminating with only half of its writes done (e.g., a subtraction from credit but no corresponding addition to debit), or a DBMS that violates durability by an incomplete recovery from a crashed history that would replay not all of the writes of a committed transaction. In both cases, semantic inconsistencies (e.g., a faulty balance of accounts, or an incomplete and thus potentially inconsistent state) are likely to happen, too. In general, each of A, C, D is complicated by concurrency, i.e., by the need to cater for I. Conversely, I is complicated whenever A, C or D has to be catered for by the DBMS as well.

3.2 Isolation or the Transparency of Concurrency

The I of $ACID$ is also called "concurrency transparency" [56], i.e., the designer or programmer or user of a transaction T does not have to worry about possibly concurrent executions of T, and the output of T in such executions is the same as if T had been executed alone, so that users and applications do not take notice of concurrency.

As already indicated in 3.1, a history H of transactions is *concurrency-transparent*, i.e., satisfies I, if it has the same effect as (or, is equivalent to) a solitary one-by-one execution of the transactions in H. To formalize this notion, it is necessary to precisely define what is "the effect of", i.e., the equivalence relation between, concurrent and sequential histoires.

Usually, each of the two most well-known variants of serializability (called conflict-serializability and, resp., view serializability [10]) is taken to ensure the sameness of effects or execution equivalence, thus guaranteeing the transparency of concurrency. Yet, serializability is only a sufficient, though not a necessary condition for isolation. And, similar to serializability, also isolation is neither sufficient nor necessary for transaction consistency. Since serializability is not always necessary for isolation, there are several similar, more or less exigent definitions of serializability [57], as well as a large amount of various weakenings, e.g., [30] [47], each of which determines its own degree or modality of isolation [6] [39] [8]. In fact, each weakening of serializability may debilitate isolation and hence transaction consistency, i.e., (*) above may no longer hold.

Instead of discussing specific definitions of serializability, we leave it here with the intuition that, for each of them, transaction designers, application programmers and users are supposed to be not bothered by concurrency, whenever transactions are guaranteed to always be executed serializably by the DBMS.

3.3 Consistency of Final and Commited States

The C of $ACID$ means semantic consistency in the context of concurrency. Independent of the given variant of serializability, there are two significantly different notions of semantic consistency of concurrent transactions in a history H: either each transaction's commit in H contributes to a state that satisfies all constraints – we call that *committed state consistency* –, or only the final state at the end of H is required to satisfy integrity – we call that *final-state consistency*. This difference had been made in [48], and was further discussed in [25].

To deal with committed-state consistency is more difficult than with final-state consistency, since the committed state at the end of any transaction T, except the one that commits last, in any history H, is not necessarily *quiescent*, i.e., is not a committed state at any time at which no transaction is in course.

In [10], a state of a database is given by "the values of the data items at any one time", while a committed state is defined with respect to some execution, "to be the state in which each data item contains its last committed value". Since non-committed concurrent transactions may be in course at any time, the committed state of a transaction T may never materialize physically at any one time. However, if all non-committed values of a transaction are protected from being accessed by other transactions, then each transaction only "sees" (parts of) committed input states. In general, the difficulties of pinpointing non-quiescent states that are committed or "seen" by users or applications tend to increase with the degree of relaxing serializability, and so do the intricacies of characterizing the consistency of such states.

Many papers about concurrent transaction processing only deal with final state consistency, and ignore what may happen with the states reached at commit time of transactions that terminate before a history comes to its end. However, transactions usually are issued without concern for the potential concurrency of their execution, and tend to have a vital interest in the consistency of their own individual outcome, rather than in the consistency of the state reached after the execution of all transactions that accidentally have been running concurrently. Hence, committed-state consistency is at least as relevant as final-state consistency, if not more.

3.4 Atomicity and Durability

The A, i.e., atomicity (also often spelled out adroitly as 'all or nothing') means that each transaction terminates either by comitting all of its updates, or by aborting without leaving behind any changes. Atomicity is a fairly straightforward standard in centralized database systems [27] [42], even if there are multiple, possibly remote users. We come back on atomicity in Sections 4 and 5.

The D, i.e., durability, has to do with the persistence of stored data beyond terminated transactions, user sessions and application program runs, and with the recovery from failures. While durability is essential for the reliability of database systems, we deliberately exclude it from further discussion in this paper, except to mention that properties of histories that define or ensure recoverability overlap but do not coincide with serializability, as pictured nicely in [10], pages 36 and 46; in particular, technologies that cater for recovery also may be beneficial for atomicity and isolation, and vice-versa.

4 Distribution Consistency and Replication Consistency

Distributed consistency means transaction consistency in distributed databases, where distribution is transparent to the designers, programmers, users and applications of transactions. (A more refined characterization of transparency is given in [56].) Similarly, replication consistency means distribution consistency in replicated databases, i.e., replication does not harm the transparency of distribution and concurrency.

For obtaining such transparency, some amount of system-level communication between the server nodes of a distributed database network is due, since data items must be accessible from each node but are either not stored at each one, or are replicated at several nodes, in which case changes of their values must be synchronized or at least coordinated to some extent.

The communication needed for data access and node coordination may suffer network latency, transmission delay and failures of nodes or network links. To cope with that, special attention has to be paid to the coordiation of concurrent accesses to replicated data items and of the atomic commit or abort of transactions. The ensurance that all commit and abort actions are carried out "consistently" is characterized in [10] as "the only non-trivial problem" caused by possible failures in distributed database systems without replication. Here, atomic commitment is a property for achieving distribution consistency. Together with transaction consistency, it is sufficient for distribution consistency. Thus, the latter can be schematically described as follows:

(**) transaction consistency + atomic commit ⇒ distribution consistency

Depending on how it is realized, atomic commitment is not categorically orthogonal to transaction consistency. For instance, if it is realized as a two- or three-phase commit [10], then the respective protocol already covers part of the work necessary to ensure isolation.

In replicated databases, the situation is further complicated by the need to make replication transparent to designers and programmers of transactions, as well as to human and programmed agents that use transactions. Thus, not only the actions of commit and abort, but also some or all actions that read or write data need to be coordinated transparently. That is usually achieved by some system protocol [28,45]. In analogy to serializability properties as mentioned in Section 3, the one-copy serializability property (1SR) [10] is a sufficient, though not a necessary property for obtaining replication consistency. That is described schematically as follows.

(***) distribution consistency + 1SR protocol ⇒ replication consistency

Usually, 1SR protocols are not orthogonal to distribution consistency. Typically, such protocols are meant to cater for part or all of distribution consistency, and also part or all of recoverability [1] [29] [2].

5 Eventual Consistency

A striking example of the babel (and sometimes babble) around 'consistency' is the ongoing discussion about 'eventual consistency' in distributed systems. In databases, it is a form of 'lazy' ('optimistic') replication consistency [49], which weakens the guarantees made by serializability, born out of the urge to scale. Eventual consistency is often mixed up with related but different issues such as availability, semantic consistency, transaction consistency, atomicity, recoverability or other consistency aspects associated to concurrency, shared memory coherence, distribution and cloud computing; see, e.g., [13] [58] [37] [63] [7] [41] [9] [3] [32].

According to [47], eventual consistency requires that replicated copies are consistent at certain times – usually in states that are quiescent for a sufficiently long period of time (some milliseconds may suffice, but maybe more) – but may be inconsistent in the interim intervals. In other words, eventual consistency means that violations of replication consistency will disappear after some indefinite delay, usually when states that are quiescent for a sufficiently long period of time are reached.

Stronger variants of eventual consistency require each violation of replication consistency to be repaired after some indefinite time, but before any write access to violated data items. Some even require that inconsistent values should be repaired by the time they are accessed, no matter if read or written, and if that is not feasible, replication inconsistency must be repaired asynchronously (typically by some compensating transactions), i.e., any consistency guarantee may be suspended indefinitely.

In summary, we can say that, in databases, eventual consistency is a compromised form of replication consistency, at the cost of strict consistency requirements. So, the question is, which kind of consistency is compromised. The answer is that each of the four kinds of consistency as addressed in Sections 2–4 can be violated in eventually consistent databases. However, what is violated in the first place by eventual consistency is the "consistency" of atomic commitment, as broached in Section 4.

Semantic consistency, transaction consistency, distribution consistency and replication consistency as characterized in Sections 2–4 are defined for whole database states, in terms of the satisfaction of integrity constraints in those states. As opposed to that, atomic commit consistency is defined for individual data items. Since eventual consistency does not comply with the strict atomic commit requirements of the *1SR* property of replication consistency, also distribution consistency, transaction consistency and semantic consistency guarantees are at risk.

Instead of requiring a coordinated order of committed updates, eventual consistency tolerates a local and immediate commit at the node where a transaction is executed. Later, the locally committed updates are propagated to the remaining database replicas, in a lazy FIFO way. Lazy propagation may be the source of consistency conflicts among concurrent transactions. Such conflicts can be dealt with by one of the two following general approaches [49].

– Each data item has a manager node. That manager uses a deterministic criterion for deciding which will be the surviving update value in case of conflicts, using compensating transactions in nodes where those values were not the latest ones being applied. That usually entails the *lost update* phenomenon, but ensures eventual value convergence in all copies.

  This solution may demand that all data items accessed by each of the concurrent transactions share the same manager node. That demand may be difficult to maintain in general.

– *Semantic scheduling.* Suppose that the operations in each transaction of a given history is commutative, and that all transactions are applied in each replica node. Then, the value of the copies of each data item will converge.

  In this case, the database system should ensure that all updates are eventually propagated to every node of the distributed network.

The data-item-based concept of atomic commit consistency is the C in the widely discussed CAP theorem [31] [14] [1]. According to [36], the C of CAP is a special case of strict serializability, where transactions are restricted to consist of a single operation applied to a single object.

Originally, CAP had not been formulated particularly for databases, but for networked shared data systems in general. According to [14], CAP says that a distributed system cannot have at a time the three properties of "consistency" (all nodes see the same data at the same time), availability (each request receives a response of success or failure) and partition tolerance (system continues operation despite loss of network connectivity).

In large-scale distributed systems (e.g., data-related cloud services deployed in multiple datacenters), network partitions may appear. In order to maintain system responsiveness to users, such systems prioritize availability and partition tolerance (i.e., A and P) over the C of CAP. In other words, the C property is being sacrificed, and that is one of the reasons for using eventual consistency instead of "strong" replication consistency in such kind of systems.

6 Partial Consistency

Partial consistency means that a given committed state may violate some constraint. Traditionally, a state that violates integrity is called inconsistent, but, arguably, calling it partially consistent is more adequate, being more suggestive of the positive potential of such states inspite of their compromised integrity, than the negative connotations associated to inconsistency.

By the usual understanding of semantic and transactional consistency, inconsistency should definitely be avoided, because inconsistencies may be severely harmful. In distributed and replicated databases, transitory inconsistency is accepted as inevitable, but even the eventual consistency paradigm insists on a convergence of the states seen by distributed users and applications toward consistency. In general, inconsistency is taken to be uncontrollable by means of classical logic, due to its *ex falso quodlibet* rule which invalidates each and every answer given to any query by an inconsistent database.

Classical logic is the acknowledged fundament of database theory and practice. Hence, inconsistency is ill-reputed, if not considered monstrous. On the other hand, inconsistency is ubiquitous in practice, to the extent that an insistence on total consistency is illusory. Moreover, practical experience shows that the majority of answers given by database systems that contain some inconsistencies are not nonsensical but valuable.

In fact, all kinds of consistency violations may easily manifest themselves in databases. For instance, legacy data that are not checked for compliance with newly introduced constraints may violate them. Or, updates that directly violate what is required by some integrity constraint will not be rejected whenever integrity control is switched off for boosting performance. Or, transaction consistency is violated by running some transaction that is not programmed according to the rules in concurrence with other transactions. Or, distributed consistency is violated by a prolongued laziness of protocols that lead to the violation of some constraint. Or, replication consistency is violated by giving priority to availability requirements and thereby weakening semantic consistency guarantees.

The predictions made by conventional transaction processing theory for transaction consistency, and hence also for distribution and replication consistency of committed database states, all depend on the fulfillment of the promise of solitary consistency preservation of each transaction. In other words, the control of transaction consistency is not in the hands of any single transaction programmer or user, but is a collective achievement of all authors of the transactions that accidentally run concurrently in the same history. However, it is naïve to trust that each transaction is written such that each of its solitary executions will preserve consistency. After all, by (*) as described in Section 3, none of the transactions in any history H is guaranteed to produce consistent outcome if there is just a single transaction in H such that a solitary execution of it fails to preserve integrity. Then, (*) does not make any consistency guarantee at all, not even for those transactions in H the solitary executions of which are perfectly consistency-preserving. Or, to put it differently: Would you bet that each transaction, programmed by a possibly unkown colleague, maybe before the latest changes in the application, that happens to run concurrently with your own (no matter how carefully written) transaction, is correctly taking into account all integrity constraints (including those that you possibly might not even know of)? If not, then the usual transaction consistency guarantees are not for you.

However, integrity violations are losing large portions of their theoretical sting, by recent advances in database research. Already since early last century, the absolute intolerance of inconsistency in logic had been questioned [12], and the ramifications of that movement have arrived in database research [18]. Although inconsistency thereby has not lost all of its potential calamities, logically sound ways to work consistently in inconsistent databases have been devised [11] [25] [24] [19] [23], so that database inconsistency has become tolerable, not only pragmatically speaking, but also from an austere theoretical point of view.

The theories presented in the preceding references provide a logical justification of database reasoning in the presence of semantic inconsistency, no matter if inconsistency has resulted from direct violations of constraints by updates, or because of any failures of controlling concurrency, distribution or replication. Concretely, [11] contains articles that describe how consistent answers to queries in inconsistent databases can be computed. In [24], it is shown that conventional integrity checking methods can be soundly applied also in inconsistent databases, even if their integrity theories are not satisfiable by any state whatsover. As shown in [23] [20], integrity violation becomes controllable and repairable by quantifying it with inconsistency measures [33]. In [19] [22], it is shown that query answers the causes of which are independent of any causes of integrity violation have the same integrity as answers in perfectly consistent databases. The authors of [25] outline how concurrent transaction consistency guarantees can be extended to inconsistent database states and histories containing transactions that do not comply with the solitary integrity preservation requirement of (*). We expect that such theories will be further developed toward a logically sound processing of transactions in situations where consistency is harmed by abandoned precautions in terms of eventual consistency, including applications of big data processing.

7 Conclusion

For clarifying the meaning of the frequently used technical term 'consistency' in the sense of stored data quality, we have characterized semantic consistency, transactional consistency, distribution consistency, replication consistency, eventual consistency and partial consistency. Semantic consistency means the satisfaction of all integrity constraints associated to a given database. Transaction consistency means concurrency-transparent semantic consistency. Distribution consistency means distribution-transparent transaction consistency. Replication consistency means replication-transparent distribution concurrency. Eventual consistency means lazy replication consistency. Partial consistency means that not necessarily all integrity constraints are satisfied.

Partial consistency is more general than all forms of consistency addressed in Sections 2–5: the latter all are oriented toward the ideal of total consistency (even though eventual consistency only promises to reach total consistency after an indefinite time), while partial consistency is prepared to always work consistenly in the presence of inconsistency, no matter if integrity violations will ever disappear or not.

The preceding lineage of database consistency that ascends from the total satisfaction of constraints for semantic consistency to a relaxation of integrity in the sense of partial consistency can be reversed to a description that descends from user-friedly partial consistency to the satisfaction of schema-level constraints, as follows. Integrity violations that are possible with partial consistency and eventual consistency are tolerable as long as they do not interfere with computing answers to queries. From a human or programmed agent's point of view, replication consistency is the same as distributed consistency, which, from the same point of view, is the same as transaction consistency, i.e., concurrency-transparent semantic consistency preservation.

We have recalled that serializability entails isolation, and also that neither serializability nor isolation alone is neither necessary nor sufficient for semantic consistency, nor for transaction consistency. Moreover, we have scrutinized the theorem that serializability in conjunction with solitary integrity preservation by all transactions in a history is sufficient (though not necessary) for transacion consistency. We have argued that the applicability of this theorem is severely limited in two ways. Firstly, total semantic consistency, and hence totally consistent states before or after the execution of transactions, are rarely given in practice, at least not in large (let alone in big) databases. Secondly, to trust in that theorem is very risky, since it actually makes no consistency guarantees unless each contingently concurrent transaction is bug-free in the sense that it will never violate integrity when run solitarily.

We have introduced the notion of partial consistency, which serves to alleviate the deficiencies of the theorem that serializability plus solitary consistency preservation entail transaction consistency. More generally, partial consistency is meant to replace the notion of inconsisteny as the natural complement of consistency. The main theoretical advantage of partial consistency over inconsistency is that classical logic throws up whenever inconsistency is encountered, while partial consistency provides a *modus vivendi* with integrity violations. The main practical advantage of partial consistency is that the consistency guarantees described schematically by (*), (**), (***) in Sections 3 and 4 are useless in databases that are not totally consistent, but become useful when relaxing the requirement of total consistency by admitting partial consistency. The main challenge of partial consistency is to apply it systematically for transaction processing in modern database systems such as column store, main-memory, NoSQL ('not only SQL') and NewSQL architectures, and to use it for reasoning with big data.

In this context, it is useful to recall that partial consistency generalizes eventual consistency (by not insisting on a convergence to total consistency), while its applications as mentioned in Section 6 (consistent query answering, inconsistency-tolerant integrity checking and repairing, inconsistency-tolerant concurrency control) do not at all forfeit strong consistency guarantees. In a similar spirit, eventual consistency is fortified in [4] by explicitly stating integrity constraints ("invariants" that define consistency conditions for a given application) as logical properties that have to hold in each state of a history ("a given set of transactions"). Thus, eventual consistency is directly linked back to semantic consistency, so that a reliable form of eventual consistency can be enforced by using inconsistency-tolerant integrity checking [24], which is an application of partial consistency. Future work may further explore this idea.

More work lies ahead also for a clarification of the meaning of 'consistency' in the fields of data science and software engineering, where 'consistency' is subsumed under 'quality', which in fact is a wider and even more fuzzy term than 'consistency'. Thus, it is a challenge to embark on a terminological study of 'data quality', similar to what we have done in this paper for 'consistency'.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. Computer **45**(2), 37–42 (2012)
2. Bailis, P. (2015). http://www.bailis.org/blog/
3. Bailis, P., Ghodsi, A.: Eventual consistency today: limitations, extensions, and beyond. ACM Queue, **11**(3) (2013)
4. Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguica, N., Najafzadeh, M., Shapiro, M.: Putting consistency back into eventual consistency. In: 10th EuroSys. ACM (2015). http://dl.acm.org/citation.cfm?doid=2741948.2741972

5. Beeri, C., Bernstein, P., Goodman, N.: A sophisticate's introduction to database normalization theory. In: VLDB, pp. 113–124 (1978)
6. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. SIGMoD Record **24**(2), 1–10 (1995)
7. Bermbach, D., Tai, S.: Eventual consistency: how soon is eventual? In: 6th MW4SOC. ACM (2011)
8. Bernabé-Gisbert, J., Muñoz-Escoí, F.: Supporting multiple isolation levels in replicated environments. Data & Knowledge Engineering **7980**, 1–16 (2012)
9. Bernstein, P., Das, S.. Rethinking eventual consistency. In: SIGMOD 2013, pp. 923–928. ACM (2013)
10. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
11. Bertossi, L., Hunter, A., Schaub, T.: Inconsistency Tolerance. In: Bertossi, L., Hunter, A., Schaub, T. (eds.) Inconsistency Tolerance. LNCS, vol. 3300, pp. 1–14. Springer, Heidelberg (2005)
12. Bobenrieth, A.: Inconsistencias por qué no? Un estudio filosófico sobre la lógica paraconsistente. Premios Nacionales Colcultura. Tercer Mundo Editores. Magister Thesis, Universidad de los Andes, Santafé de Bogotá, Columbia (1995)
13. Bosneag, A.-M., Brockmeyer, M.: A formal model for eventual consistency semantics. In: PDCS 2002, pp. 204–209. IASTED (2001)
14. Browne, J.: Brewer's cap theorem (2009). http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
15. Cong, G., Fan, W., Geerts, F., Jia, X., Ma, S.: Improving data quality: consistency and accuracy. In: Proc. 33rd VLDB, pp. 315–326. ACM (2007)
16. Dechter, R., van Beek, P.: Local and global relational consistency. Theor. Comput. Sci. **173**(1), 283–308 (1997)
17. Decker, H.: Translating advanced integrity checking technology to SQL. In: Doorn, J., Rivero, L. (eds.) Database integrity: challenges and solutions, pp. 203–249. Idea Group (2002)
18. Decker, H.: Historical and computational aspects of paraconsistency in view of the logic foundation of databases. In: Bertossi, L., Katona, G.O.H., Schewe, K.-D., Thalheim, B. (eds.) Semantics in Databases 2001. LNCS, vol. 2582, pp. 63–81. Springer, Heidelberg (2003)
19. Decker, H.: Answers that have integrity. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2010. LNCS, vol. 6834, pp. 54–72. Springer, Heidelberg (2011)
20. Decker, H.: New measures for maintaining the quality of databases. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part IV. LNCS, vol. 7336, pp. 170–185. Springer, Heidelberg (2012)
21. Decker, H.: A pragmatic approach to model, measure and maintain the quality of information in databases (2012). www.iti.upv.es/~hendrik/papers/ahrc-workshop_quality-of-data.pdf, www.iti.upv.es/~hendrik/papers/ahrc-workshop_quality-of-data_comments.pdf. Slides and comments presented at the Workshop on Information Quality. Univ, Hertfordshire, UK
22. Decker, H.: Answers that have quality. In: Murgante, B., Misra, S., Carlini, M., Torre, C.M., Nguyen, H.-Q., Taniar, D., Apduhan, B.O., Gervasi, O. (eds.) ICCSA 2013, Part II. LNCS, vol. 7972, pp. 543–558. Springer, Heidelberg (2013)
23. Decker, H.: Measure-based inconsistency-tolerant maintenance of database integrity. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2013. LNCS, vol. 7693, pp. 149–173. Springer, Heidelberg (2013)

24. Decker, H., Martinenghi, D.: Inconsistency-tolerant integrity checking. IEEE Transactions of Knowledge and Data Engineering **23**(2), 218–234 (2011)
25. Decker, H., Muñoz-Escoí, F.D.: Revisiting and improving a result on integrity preservation by concurrent transactions. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2010. LNCS, vol. 6428, pp. 297–306. Springer, Heidelberg (2010)
26. Dong, X.L., Berti-Equille, L., Srivastava, D.: Data fusion: resolving conflicts from multiple sources (2015). http://arxiv.org/abs/1503.00310
27. Eswaran, K., Gray, J., Lorie, R., Traiger, I.: The notions of consistency and predicate locks in a database system. CACM **19**(11), 624–633 (1976)
28. Muñoz-Escoí, F.D., Ruiz-Fuertes, M.I., Decker, H., Armendáriz-Íñigo, J.E., de Mendívil, J.R.G.: Extending middleware protocols for database replication with integrity support. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 607–624. Springer, Heidelberg (2008)
29. Fekete, A.: Consistency models for replicated data. In: Encyclopedia of Database Systems, pp. 450–451. Springer (2009)
30. Fekete, A., Gupta, D., Lynch, V., Luchangco, N., Shvartsman, A.: Eventually-serializable data services. In: 15th PoDC, pp. 300–309. ACM (1996)
31. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002)
32. Golab, W., Rahman, M., Auyoung, A., Keeton, K., Li, X.: Eventually consistent: Not what you were expecting? ACM Queue, **12**(1) (2014)
33. Grant, J., Hunter, A.: Measuring inconsistency in knowledgebases. Journal of Intelligent Information Systems **27**(2), 159–184 (2006)
34. Gray, J., Lorie, R., Putzolu, G., Traiger, I.: Granularity of locks and degrees of consistency in a shared data base. In: Nijssen, G. (ed.) Modelling in Data Base Management Systems. North Holland (1976)
35. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. Computing Surveys **15**(4), 287–317 (1983)
36. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. TOPLAS **12**(3), 463–492 (1990)
37. R. Ho. Design pattern for eventual consistency (2009). http://horicky.blogspot.com.es/2009/01/design-pattern-for-eventual-consistency.html
38. Ikeda, R., Park, H., Widom, J.: Provenance for generalized map and reduce workflows. In: CIDR (2011)
39. Kempster, T., Stirling, C., Thanisch, P.: Diluting acid. SIGMoD Record **28**(4), 17–23 (1999)
40. Li, X., Dong, X.L., Meng, W., Srivastava, D.: Truth finding on the deep web: Is the problem solved? VLDB Endowment **6**(2), 97–108 (2012)
41. Lloyd, W., Freedman, M., Kaminsky, M., Andersen, D.: Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: 23rd SOPS, pp. 401–416 (2011)
42. Lomet, D.: Transactions: from local atomicity to atomicity in the cloud. In: Jones, C.B., Lloyd, J.L. (eds.) Dependable and Historic Computing. LNCS, vol. 6875, pp. 38–52. Springer, Heidelberg (2011)
43. Monge, P., Contractor, N.: Theory of Communication Networks. Oxford University Press (2003)
44. Nicolas, J.-M.: Logic for improving integrity checking in relational data bases. Acta Informatica **18**, 227–253 (1982)
45. Muñoz-Escoí, F.D., Irún, L., H. Decker: Database replication protocols. In: Encyclopedia of Database Technologies and Applications, pp. 153–157. IGI Global (2005)

46. Oracle: Constraints. http://docs.oracle.com/cd/B19306_01/server.102/b14223/constra.htm (May 1, 2015)
47. Ouzzani, M., Medjahed, B., Elmagarmid, A.: Correctness criteria beyond serializability. In: Encyclopedia of Database Systems, pp. 501–506. Springer (2009)
48. Rosenkrantz, D., Stearns, R., Lewis, P.: Consistency and serializability in concurrent datanbase systems. SIAM J. Comput. **13**(3), 508–530 (1984)
49. Saito, Y., Shapiro, M.: Optimistic replication. JACM **37**(1), 42–81 (2005)
50. Sandhu, R.: On five definitions of data integrity. In: Proc. IFIP WG11.3 Workshop on Database Security, pp. 257–267. North-Holland (1994)
51. Simmons, G.: Contemporary Cryptology: The Science of Information Integrity. IEEE Press (1992)
52. Sivathanu, G., Wright, C., Zadok, E.: Ensuring data integrity in storage: techniques and applications. In: Proc. 12th Conf. on Computer and Communications Security, p. 26. ACM (2005)
53. Svanks, M.: Integrity analysis: Methods for automating data quality assurance. Information and Software Technology **30**(10), 595–605 (1988)
54. Technet, M.: Data integrity. https://technet.microsoft.com/en-us/library/aa933058 (May 1, 2015)
55. Terry, D.: Replicated data consistency explained through baseball. Technical report, Microsoft. MSR Technical Report (2011)
56. Traiger, I., Gray, J., Galtieri, C., Lindsay, B.: Transactions and consistency in distributed database systems. ACM Trans. Database Syst. **7**(3), 323–342 (1982)
57. Vidyasankar, K.: Serializability. In: Encyclopedia of Database Systems, pp. 2626–2632. Springer (2009)
58. Vogels, W.: Eventually consistent (2007). http://www.allthingsdistributed.com/2007/12/eventually_consistent.html. Other versions in ACM Queue **6**(6), 14–19. http://queue.acm.org/detail.cfm?id=1466448 (2008) and CACM **52**(1), 40–44 (2009)
59. Wikipedia: Consistency model. http://en.wikipedia.org/wiki/Consistency_model (May 1, 2015)
60. Wikipedia: Data integrity. http://en.wikipedia.org/wiki/Data_integrity (May 1, 2015)
61. Wikipedia: Data quality. http://en.wikipedia.org/wiki/Data_quality (May 1, 2015)
62. Yin, X., Han, J., Yu, P.: Truth discovery with multiple conflicting information providers on the web. IEEE Transactions of Knowledge and Data Engineering **20**(6), 796–808 (2008)
63. Young, G.: Quick thoughts on eventual consistency (2010). http://codebetter.com/gregyoung/2010/04/14/quick-thoughts-on-eventual-consistency/ (May 1, 2015)