I/O States as Seen by Concurrent Transactions

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

Instituto Tecnológico de Informática, 46071 Valencia, Spain
hendrik@iti.upv.es, fmunyoz@iti.upv.es

Abstract. If not all resources accessed by a database transaction are protected from being accessed by other concurrent transactions, then the state "seen" by the transaction is not necessarily identical to any committed state, nor to any snapshot of the current contents of the stored data. For the theory of concurrent database transactions as well as for all database applications that involve concurrency, it is important to be precise about the states with which theories or applications are dealing. Based on a non-standard notion of data resource, we propose a formalization of committed states, snapshots and I/O states that are 'seen' by concurrent transactions. We intend to apply our concept of states to an application of inconsistency-tolerant integrity checking for concurrent transactions.

1 Introduction

For a database transaction T executed concurrently with other transactions in some history, the problem discussed in this paper is to determine which is the state 'seen' by T as its 'input' (i.e., the state from which T first reads data), and which is the state that is partially determined by its 'output' (i.e., by the writeset of T).

For convenience, we shall from now on simply speak of 'transactions', although we exclusively deal with database transactions. On the other hand, it can be supposed that the issues discussed in this paper may bear relevance not only for transactions in databases, but also for transactions in other applications with concurrent processes.

The traditional approach for avoiding well-known anomalies of concurrent transactions (dirty reads, lost updates, unrepeatable reads) is to protect all resources accessed by a transaction T. Protection usually is achieved by locking accessed resources from beginning to end of T. That way, a sufficient isolation level of T is obtained. By isolating concurrent transactions, the history of their execution becomes *serializable.* Serializability means that concurrently executed transactions have the effect as of being executed in a one-by-one sequence, i.e., any interference between them that could be caused by their concurrency is avoided.

However, for many applications, locking off resources means to block them unnecessarily. For instance, read transactions that are only interested in the

[*] Partially supported by FEDER and the Spanish MEC grant TIN2006-14738-C02-01

current value of a locked data resource may be unduly delayed. Fortunately, however, concurrent transactions of many applications may remain serializable with suitably relaxed locking policies.

Depending on the application, even the serializability requirement can be relaxed sometimes, without incurring unwanted consequences. For instance, most commercial database systems only offer 'snapshot isolation' for concurrent transactions. That has the advantages of being easily implemented and of providing faster throughput than total isolation, while providing a consistent view of some state of the database to each transaction. The disadvantage is that the state seen by concurrent transactions that run with snapshot isolation is not necessarily the most recently committed one.

In theory and practice, it is often necessary to be precise about the states related to concurrent transactions, e.g., for reasoning about their content, their integrity, their accessibility, their chronology and their coordination. Thus, it should be useful to be able to clearly distinguish between states that are snapshots (i.e., the values of the data items at any one point of time), states that are committed or not, states that are seen as input by transactions and states that are determined by the output of transactions.

In the literature, it is common to discuss 'snapshots', 'committed states', and states 'seen' by transactions. Also, state-like phenomena like 'spheres of control', 'checkpoints', 'savepoints' are addressed frequently. However, precisely differentiating definitions of different kinds of states are mostly missing. This paper, in particular its section 2, is an attempt to close that gap.

In section 3, we briefly address related work. In section 4, we conclude.

We assume a basic familiarity with databases in general [19], and in particular with concurrent database transactions [2] [13].

2 Revisiting established notions wrt. concurrency

We revisit established notions and formalizations traditionally used in the literature on concurrent transactions in databases.

Drawing from [9, 2, 16], we introduce in 2.1 - 2.3, revised notions of resource (a.k.a. data item), snapshot, committed state, i/o 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 'objects', nor on snapshots, nor on committed states, but on logical truth values of relational tuples in i/o states.

2.1 Databases, Resources, States, Snapshots, Transactions

A *database* is a *schema* S of relational table structures over some language L_S [19]. Throughout the remainder, we assume that some arbitrary schema S is given.

A *resource* is a unit of storable information. As in [16], the only resources considered in this paper are the elements of the Herbrand base of L_S.

A *database state* (in short, *state*) is a mapping from resources to $\{true, false\}$. A state is *partial* if the mapping is partial.

Throughout, states are denoted by symbols D and convenient adornments thereof. The value of a resource r in a state D is denoted by $D(r)$.

For a state D and a resource r, we say that the value of r is *known* (*true* or *false*) in D if $D(r) = true$ or, resp., $D(r) = false$. Else, r is *unknown* in D.

Note that the definition of 'state' above abstracts away from any aspect of time, sequence or dynamics. As opposed to that, the authors of [2] define that "the values of the data items at any one time comprise the state of the database." We call such states *snapshots*. Examples of snapshots and other kind of states are given later, in 2.2.

States are changed over time by actions, to be defined next.

An *action* is an operation that *acts on* precisely one resource, except begin and end, as defined below, which act on no or, resp., possibly many resources.

Each action is executed atomically at one point of time. (We assume a sufficiently fine-grained unbounded sequence of discrete time points at which actions are executed.)

Typical actions are *read* and *write*. Two actions *conflict* if both act on the same resource and one of them is a write.

A *transaction* is a finite set of actions that is partically ordered over time. Each transaction T consists of precisely one *begin*, precisely one *end* of the form *commit* or *abort*, and a finite set of *access* actions, each of the form *read*(r) or *write*(r), where r stands for a resource. The begin (end) of T is earlier (resp., later) than each access of T. Conflicting actions in T are never executed at the same time.

We may speak elliptically of a transaction T when in fact we have an execution of T in mind.

Distinguished examples of snapshots are the states at the time a transaction T begins and, resp., ends, which we denote by D^b_T and, resp., D^e_T.

To *read* a resource r means to query if r is *true* or *false* of the current 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. If T writes r and commits at some point of time t, we also say that T *acts on* and *commits* r *at time* t (t may remain unmentioned). Thereby, T confirms its last write to 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, $T \in \mathbb{C}_T$.

2.2 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. (We assume a sufficiently fine-grained unbounded sequence of discrete points of time that underlies all executions of actions.)

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.

Note that, by this definition, histories are complete in the sense of [2], i.e., they contain each action of each $T \in \mathbb{T}$.

A *long history* be informally defined as a history that spans over an extended period of time (typically, several hours, possibly days). Note that a long history is not the same as a long transaction, nor a collection of long transactions. Rather, all (or most) of the transactions in a long history may be of regular, relatively short duration. Thus, the typical case of a long history is that it involves a large number of transactions, the beginnings of ends of which partially overlap with each other. Long histories typically occur in OLTP applications like on-line seat reservation systems.

For convenience, we say that a transaction T *is in* H if H is a history of a set of transactions $\mathbb{T}$ and $T \in \mathbb{T}$.

In general, we assume that histories are *inclusive*, i.e., for each transaction T in H, also each transaction $T' \in \mathbb{C}_T$ is in H; otherwise, scheduling may not consider all possible conflicts. Thus, histories may be arbitrarily long, to the extent that the beginning of a long history may be forgotten or unknown, and its end may be out of sight.

Distinguished snapshots at which no access takes place are the states at the time of the earliest begin and the latest end in H, denoted by D^b_H and, resp., D^e_H. D^e_H is 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 that has been committed most recently by some transaction 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 $[t_c, t]$.

In concurrency theory, there are states that are not necessarily snapshots, i.e., the values of resources do not necessarily correspond to the same point of time. Examples of such notions of states are, e.g., the 'states seen by transactions' [9], 'global states' [10], and 'distributed snapshots' [4].

Another example is the class of *committed states*. It is defined by the committed value of each resource at some time t in H. Note that, in general, the committed state at time t is different from the snapshot at t.

Another example of states that are not necessarily snapshots is the class of *i/o states*. They are partial, since transactions usually 'see' (access) only part of the database. In 2.3, we use i/o states also for characterizing serializability.

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

in the *output state* D^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 D^i_T and D^o_T remain unknown.

Clearly, $D^i_T \subseteq D^b_T$ and $D^o_T \subseteq D^e_T$ if T is executed in isolation. I/o states are not necessarily snapshots, since they may not exist at any fixed point of time. In particular, D^i_T and D^o_T may be different from D^b_T or, resp., D^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, D^i_T and D^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 $D^i_{T1} = D^o_{T1}$, which is not a committed state at any time of H. Yet, in general, D^i_T and D^o_T are the first and, resp., the last state 'seen by' T.

I/o states facilitate the modeling of long histories in 'non-stop' DBSs for 24/7 applications, where the initial or terminal committed states at the time of the begin or the end of histories may not exist or may be out of sight. Also the modeling of histories with relaxed isolation requirements is easier with i/o states, since they do not necessarily coincide with committed states.

2.3 Serializability

The serializability of a history H (usually taken care of transparently by a DBS 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 (and thus also for T', T), 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 database" [2]. Anomalies are not the only possible cause of integrity violation, and integrity sometimes is preserved in spite of anomalies. 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 HS of the transactions in H such that $D^b_{HS} = D^b_H$. For example, the history of *T0*, *T1*, *T2* in 2.2 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 [9, 2].

3 Related work

Related work by [4] [10] concerned with difficulties of defining states in the context of concurrency and distribution has already been mentioned in 2.2.

The rest of this section is written in anticipation of an application of the revised concepts of concurrency presented above to the field of integrity checking. All integrity checking methods that are described in the literature or are implemented in database products claim to preserve integrity satisfaction whenever the database is updated by some transaction T. Integrity satisfaction and violation is always defined with regard to some state. Hence, the following question arises: Which are the states that are claimed to satisfy integrity in databases with concurrent transactions?

In early work [12, 8, 14, 9, 1, 11], a distinction is made between integrity violations caused either by anomalies of concurrency or semantic errors. In [8, 14], concurrency is not dealt with any further. In [12, 9, 1, 11], integrity is not looked at in detail. Also in later related work, either concurrency or integrity is passed by, except in [3, 20, 16]. These papers address the particular problem of integrity satisfaction in the final state D^e_H of a history H, while ignoring the problem of integrity guarantees for states D^e_T or D^o_T of transactions $T \in H$ that do not coincide with D^e_H.

In [3], Böttcher observes that integrity checks are read-only actions without effect on other operations, possibly except abortions due to integrity violation. Some scheduling optimizations made possible by the unobtrusive nature of read actions for integrity checking are discussed in [3].

For write transactions, the parametrizable actions of which are specified ahead of their execution, Martinenghi and Christiansen describe in [16] how to augment transactions with locks and actions for simplified integrity checking, so that their serializable execution guarantees integrity preservation.

The states that are guaranteed to satisfy integrity usually are the states 'after the update', often under the premise that integrity is satisfied in the state 'before the update'. Such 'before' and 'after' states are clearly defined for transactions executed in isolation, since then they coincide with D^b_T and, resp., D^e_T. However, as soon as isolation is compromised, it is not obvious which states can be guaranteed to satisfy integrity.

Each transaction T usually is required to preserve the satisfaction of integrity in the case that $\mathbb{C}_T = \emptyset$. Then, each serializable history of concurrent transactions that comply with this requirement is guaranteed to preserve integrity from 'before' to 'after' states [9]. But, again, if serializability is compromised, then such guarantees become questionable.

Even if isolation and serializability are not compromised, there is another difficulty of applying methods for integrity preservation to concurrent transactions: the input states of transactions do not necessarily include all resources that have to be accessed by the integrity checking method. Hence, guarantees made with regard to the i/o states of transactions become questionable if the integrity constraints involve data that are not contained in the i/o states. Such situations typically arise with transactions that update a table that is in a foreign key relationship with some other table.

First steps to find answers to these questions are taken in [7].

Difficulties of identifying equivalent, i.e., mutually consistent states as seen by concurrent transactions in replicated databases are discussed in [17]. Additional difficulties of coordinating standard replication protocols with integrity checking are addressed in [18] [15].

4 Outlook and Conclusion

We have defined states 'seen by' concurrent transactions as partial states called i/o states. They typically contain unknown values.

For concurrent transactions and even more so for long histories, it is virtually impossible to guarantee totally consistent committed states at all times. For i/o states, the requirement of total consistency is even more difficult to comply with. Thus, an inconsistency-tolerant approach to integrity checking is needed. Based on i/o states and a recently introduced notion of inconsistency-tolerant integrity checking [5, 6], we have scrutinized, restated and generalized, in an extended draft of this paper, the classic result that integrity preserved in isolation is sufficient for preserving integrity concurrently. Ongoing work is concerned with a further elaboration and publication of that draft.

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. A relaxed approach to integrity and inconsistency in databases. *Proc. 13th LPAR*, LNCS vol. 4246, 287-301. Springer, 2006.
6. H. Decker, D. Martinenghi. Classifying Integrity Checking Methods with regard to Inconsistency Tolerance. *Proc. 10th PPDP*, 195-204. ACM Press, 2008.
7. H. Decker, F. D. Muñoz-Escoí. Business Rules for Concurrent e-Commerce Transactions. *Proc. 4th DEECS*, IEEE CS, to appear, 2009.
8. K. Eswaran, D. Chamberlin. Functional Specification of a Subsystem for Data Base Integrity. *Proc. 1st VLDB*, 48-68. ACM Press, 1975.
9. 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.
10. M. Fischer, N. Griffeth, N. Lynch. Global States of a Distributed System. *IEEE Trans. Software Eng.* 8(3):198-202, 1982.
11. G. Gardarin. Integrity, Consistency, Concurrency, Reliability in Distributed Database Management Systems. In C. Delobel, W. Litwin (eds), *Distributed Databases*, 335-351. North-Holland, 1980.

12. J. Gray, R. Lorie, G. Putzolu. Granularity of Locks in a Shared Data Base. *Proc. 1st VLDB*, 428-451. ACM Press, 1975.
13. J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
14. M. Hammer, D. McLeod. Semantic Integrity in a Relational Data Base System. *Proc. 1st VLDB*, 25-47. ACM Press, 1975.
15. Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, J. E. Armendáriz-Íñigo: Snapshot Isolation and Integrity Constraints in Replicated Databases. To appear in *Transactions on Database Systems*, 2009.
16. D. Martinenghi, H. Christiansen. Transaction Management with Integrity Checking. *Proc. 16th DEXA*, LNCS vol. 3588, 606-615. Springer, 2005.
17. F. D. Muñoz-Escoí, J. M. Bernabé-Gisbert, R. de Juan-Marín: Surveying Correctness Criteria in Replicated Databases with Snapshot Isolation. *Proc. 17th JCSD*, Universidad Politécnica de Valencia, 2009.
18. F. D. Muñoz-Escoí, M. I. Ruiz-Fuertes, H. Decker, J. E. Armendáriz-Íñigo, J. R. González de Mendivil. Extending Middleware Protocols for Database Replication with Integrity Support. *Proc. OTM 2008, Part I*, LNCS vol. 5331, 607-624. Springer, 2008.
19. R. Ramakrishnan, J. Gehrke. *Database Management Systems*, 3rd edition. McGraw-Hill, 2003.
20. A. Silberschatz, Z. Kedem. Consistency in Hierarchical Database Systems. *JACM* 27(1):72-80, 1980.
21. K. Vidyasankar. Serializability. To appear in L. Liu, T. Özu (eds), *Encyclopedia of Database Systems*. Springer, 2009.