Correctness Criteria Beyond Serializability

Mourad Ouzzani
Cyber Center, Purdue University
http://www.cs.purdue.edu/homes/mourad/
Brahim Medjahed
Department of Computer & Information Science, The University of Michigan - Dearborn
http://www.engin.umd.umich.edu/∼brahim
Ahmed K. Elmagarmid
Cyber Center and Department of Computer Science, Purdue University
http://www.cs.purdue.edu/homes/ake/

SYNONYMS

Concurrency Control; Preserving Database Consistency

DEFINITION

A *transaction* is a logical unit of work that includes one or more database access operations such as insertion, deletion, modification, and retrieval [8]. A *schedule* (or history) S of n transactions T_1,...,T_n is an ordering of the transactions that satisfies the following two conditions: (i) the operations of T_i (i=1,...,n) in S must occur in the same order in which they appear in T_i, and (ii) operations from T_j (j≠i) may be interleaved with T_i's operations in S. A schedule S is *serial* if for every two transactions T_i and T_j that appear in S, either all operations of T_i appear before all operations of T_j, or vice versa. Otherwise, the schedule is called *nonserial* or *concurrent*. Non-serial schedules of transactions may lead to concurrency problems such as lost update, dirty read, and unrepeatable read. For instance, the lost update problem occurs whenever two transactions, while attempting to modify a data item, both read the item's old value before either of them writes the item's new value [2].
The simplest way for controlling concurrency is to allow only serial schedules. However, with no concurrency, database systems may make poor use of their resources and hence, be inefficient, resulting in smaller transaction execution rate for example. To broaden the class of allowable transaction schedules, *serializability* has been proposed as the major correctness criterion for concurrency control [7, 11]. Serializability ensures that a concurrent schedule of transactions is equivalent to some serial schedule of the same transactions [12]. While serializability has been successfully used in traditional database applications, e.g., airline reservations and banking, it has been proven to be restrictive and hardly applicable in advanced applications such as Computer-Aided Design (CAD), Computer-Aided Manufacturing (CAM), office automation, and multidatabases. These applications introduced new requirements that either prevent the use of serializability (e.g., violation of local autonomy in multidatabases) or make the use of serializability inefficient (e.g,, long-running transactions in CAD/CAM applications). These limitations have motivated the introduction of more flexible correctness criteria that go beyond the traditional serializability.

HISTORICAL BACKGROUND

Concurrency control began appearing in database systems in the early to mid 1970s. It emerged as an active database research thrust starting from 1976 as witnessed by the early influential papers published by Eswaren et al. [5] and Gray et al. [7]. A comprehensive coverage of serializability theory has been presented in 1986

by Papadimitriou in [12]. Simply put, serializability theory is a mathematical model for proving whether or not a concurrent execution of transactions is correct. It gives precise definitions and properties that non-serial schedules of transactions must satisfy to be serializable. Equivalence between a concurrent and serial schedule of transactions is at the core of the serializability theory. Two major types of equivalence have then been defined: *conflict* and *view* equivalence. If two schedules are conflict equivalent then they are view equivalent. The converse is not generally true.

Conflict equivalence has initially been introduced by Gray et al. in 1975 [7]. A concurrent schedule of transactions is *conflict equivalent* to a serial schedule of the same transactions (and hence *conflict serializable*) if they order conflicting operations in the same way, i.e., they have the same precedence relations of conflicting operations. Two operations are *conflicting* if they are from different transactions upon the same data item, and at least one of them is `write`. If two operations conflict, their execution order matters. For instance, the value returned by a `read` operation depends on whether or not that operation precedes or follows a particular `write` operation on the same data item. Conflict serializability is tested by analyzing the acyclicity of the graph derived from the execution of the the different transactions in a schedule. This graph, called *serializability graph*, is a directed graph that models the precedence of conflicting operations in the transactions.

View equivalence has been proposed by Yannakakis in 1984 [15]. A concurrent schedule of transactions is *view equivalent* to a serial schedule of the same transactions (and hence *view serializable*) if the respective transactions in the two schedules read and write the same data values. View equivalence is based on the following two observations: (1) if each transaction reads each of its data items from the same `write`s, then all `write`s write the same value in both schedules; and (2) if the final `write` on each data item is the same in both schedules, then the final value of all data items will be the same in both schedules. View serializability is usually expensive to check. One approach is to check the acyclicity of a special graph called *polygraph*. A polygraph is a generalization of the precedence graph that takes into account all precedence constraints required by view serializability.

SCIENTIFIC FUNDAMENTALS

The limitations of the traditional serializability concept combined with the requirement of advanced database applications triggered a wave of new correctness criteria that go beyond serializability. These criteria aim at achieving one or several of the following goals: (1) accept non serializable but correct executions by exploiting the semantics of transactions, their structure, and integrity constraints (2) allow inconsistencies to appear in a controlled manner which may be acceptable for some transactions, (3) limit conflicts by creating a new version of the data for each update, and (4) treat transactions accessing more than one database, in the case of multidatabases, differently from those accessing one single database and maintain overall correctness. While a large number of correctness criteria have been presented in the literature, this entry will focus on the major criteria which had a considerable impact on the field. These criteria will be presented as described in their original versions as several of these criteria have been either extended, improved, or applied to specific contexts. Table 1 summarizes the correctness criteria outlined in this section.

Multiversion Serializability

Multiversion databases aim at increasing the degree of concurrency and providing a better system recovery. In such databases, whenever a transaction writes a data item, it creates a new version of this item instead of overwriting it. The basic idea of *multiversion serializability* [1] is that some schedules can be still seen as serializable if a read is performed on some older version of a data item instead of the newer modified version. Concurrency is increased by having transactions read older versions while other concurrent transactions are creating newer versions. There is only one type of conflict that is possible; when a transactions reads a version of a data item that was written by another transaction. The two other conflicts (write, write) and (read, write) are not possible since each write produces a new version and a data item cannot be read until it has been produced, respectively. Based on the assumption that users expect their transactions to behave as if there were just one copy of each data item, the notion of *one-copy serial* schedule is defined. A schedule is one-copy serial if for all i, j, and x, if a transaction T_j reads x from a transaction T_i, then either $i = j$ or T_i is the last transaction preceding t_j

Correctness Criterion	Basic Idea	Examples of Application Domains	Reference
Multiversion Serializability	Allows some schedules as serializable if a read is performed on some older version of a data item instead of the newer modified version.	Multiversion database systems.	[1]
Semantic Consistency	Uses semantic information about transactions to accept some non-serializable but correct schedules.	Applications that can provide some semantic knowledge	[6]
Predicatewise Serializability	Focuses on data integrity constraints.	CAD database and office information systems	[9]
Epsilon-Serializability	Allows inconsistencies to appear in a controlled manner by attaching a specification of the amount of permitted inconsistency to each transaction.	Applications that tolerate some inconsistencies	[13]
Eventual Consistency	Requires that duplicate copies are consistent at certain times but may be inconsistent in the interim intervals.	Distributed databases with replicated or interdependent data.	[14]
Quasi Serializability	Executes global transactions in a serializable way while taking into account the effect of local transactions.	Multidatabase systems.	[4]
Two-level Serializability	Ensures consistency by exploiting the nature of integrity constraints and the nature of transactions in multidatabase environments.	Multidatabase systems.	[10]

Table 1: Representative Correctness Criteria for Concurrency Control

that writes into any version of x. Hence, a schedule is defined as *one-copy serializable* (1-SR) if it is equivalent to a 1-serial schedule. 1-SR is shown to maintain correctness by proving that a multiversion schedule behaves like a serial non-multiversion schedule (there is only one version for each data item) iff the multiversion schedule is one-serializable. The one-copy serializability of a schedule can be verified by checking the acyclicity of the multiversion serialization graph of that schedule.

Semantic Consistency

Semantic consistency uses semantic information about transactions to accept some non-serializable but correct schedules [6]. To ensure that users see consistent data, the concept of *sensitive transactions* has been introduced. Sensitive transactions output only consistent data and thus must see a consistent database state. A semantically consistent schedule is one that transforms the database from a consistent state to another consistent state and where all sensitive transactions obtain a consistent view of the database with respect to the data accessed by these transactions, i.e., all data consistency constraints of the accessed data are evaluated to True. Enforcing semantic consistency requires knowledge about the application which must be provided by the user. In particular, users will need to group actions of the transactions into steps and specify which steps of a transaction of a given type can be interleaved with the steps of another type of transactions without violating consistency. Four types of semantic knowledge are defined: (1) transaction semantic types, (2) compatibility sets associated with each type, (3) division of transactions into steps, and (4) counter-steps to (semantically) compensate the effect from some of the steps executed within the transaction.

Predicatewise Serializability

Predicatewise serializability (PWSR) has been introduced as a correctness criterion for CAD database and office

information systems [9]. PWSR focuses solely on data integrity constraints. In a nutshell, if database consistency constraints can be expressed in a conjunctive normal form, a schedule is said to be PWSR if all projections of that schedule on each group of data items that share a disjunctive clause (of the conjunctive form representing the integrity constraints) are serializable. There are three different types of restrictions that must be enforced on PWSR schedules to preserve database consistency: (1) force the transactions to be of *fixed structure*, i.e., they are independent of the database state from which they execute, (2) force the schedules to be *delayed read*, i.e., a transaction T_i cannot read a data item written by a transaction T_j until after T_j has completed all of its operations, or (3) the conjuncts of the integrity constraints can be ordered in a way that no transaction reads a data item belonging to a higher numbered conjunct and writes a data item belonging to a lower numbered conjunct.

Epsilon-Serializability

Epsilon-serializability (ESR) [13] has been introduced as a generalization to serializability where a limited amount of inconsistency is permitted. The goal is to enhance concurrency by allowing some non serializable schedules. ESR introduces the notion of *epsilon transactions* (ETs) by attaching a specification of the amount of permitted inconsistency to each (standard) transaction. ESR distinguishes between transactions that contain only read operation, called query epsilon transaction or query ET, and transactions with at least one update operation, called update epsilon transaction or update ET. Query ETs may view uncommitted, possibly inconsistent, data being updated by update ETs. Thus, update ETs are seen as exporting some inconsistencies while query ETs are importing these inconsistencies. ESR aims at bounding the amount of imported and exported inconsistency for each ET. An *epsilon-serial* schedule is defined as a schedule where (i) the update ETs form a serial schedule if considered alone without the query ET and (ii) the entire schedule consisting of both query ETs and update ETs is such that the non serializable conflicts between query ETs and update ETs are less than the permitted limits specified by each ET. An epsilon-serializable schedule is one that is equivalent to an epsilon-serial schedule. If the permitted limits are set to zero, ESR corresponds to the classical notion of serializability.

Eventual Consistency

Eventual consistency has been proposed as an alternative correctness criterion for distributed databases with replicated or interdependent data [14]. This criterion is useful is several applications like mobile databases, distributed databases, and large scale distributed systems in general. Eventual consistency requires that duplicate copies are consistent at certain times but may be inconsistent in the interim intervals. The basic idea is that duplicates are allowed to diverge as long as the copies are made consistent periodically. The times where these copies are made consistent can be specified in several ways which could depend on the application, for example, at specified time intervals, when some events occur, or at some specific times. A correctness criterion that ensures eventual consistency is the *current copy serializability*. Each update occurs on a current copy and is asynchronously propagated to other replicas.

Quasi Serializability

Quasi Serializability (QSR) is a correctness criterion that has been introduced for multidatabase systems [4]. A multidatabase system allows users to access data located in multiple autonomous databases. It generally involves two kinds of transactions: (i) Local transactions that access only one database; they are usually outside the control of the multidatabase system, and (ii) global transactions that can access more than one database and are subject to control by both the multidatabase and the local databases. The basic premise is that to preserve global database consistency, global transactions should be executed in a serializable way while taking into account the effect of local transactions. The effect of local transactions appears in the form of indirect conflicts that these local transactions introduce between global transactions which may not necessarily access (conflict) the same data items. A *quasi serial* schedule is a schedule where global transactions are required to execute serially and local schedules are required to be serializable. This is in contrast to global serializability where all transactions, both local and global, need to execute in a (globally) serializable way. A global schedule is said to be quasi serializable if it is (conflict) equivalent to a quasi serial schedule. Based on this definition, a quasi serializable schedule maintains

the consistency of multidatabase systems since (1) a quasi serial schedule preserves the mutual consistency of globally replicated data items, based on the assumptions that these replicated data items are updated only by global transactions, and (2) a quasi serial schedule preserves the global transaction consistency constraints as local schedules are serializable and global transactions are executed following a schedule that is equivalent to a serial one.

Two-Level Serializability

Two-level serializability (2LSR) has been introduced to relax serializability requirements in multidatabases and allow a higher degree of concurrency while ensuring consistency [10]. Consistency is ensured by exploiting the nature of integrity constraints and the nature of transactions in multidatabase environments. A global schedule, consisting of both local and global transactions, is 2LSR if all local schedules are serializable and the projection of that schedule on global transactions is serializable. Local schedules consist of all operations, from global and local transactions, that access the same local database. Ensuring that each local schedule is serializable is already taken care of by the local database. Furthermore, ensuring that the global transactions are executed in a serializable way can be done by the global concurrency controller using any existing technique from centralized databases like the Two-phase-locking (2PL) protocol. This is possible since the global transactions are under the full control of the global transaction manager. [10] shows that under different scenarios 2LSR preserves a strong notion of correctness where the multidatabase consistency is preserved and all transactions see consistent data. These different scenarios differ depending on (i) which kind of data items, local or global, global and local transactions are reading or writing, (ii) the existence of integrity constraints between local and global data items, and (iii) whether all transaction are preserving the consistency of local databases when considered alone.

KEY APPLICATIONS

The major database applications behind the need for new correctness criteria include distributed databases, mobile databases, multidatabases, CAD/CAM applications, office automation, cooperative applications, and software development environments. All of these advanced applications introduced requirements and limitations that either prevent the use of serializability like the violation of local autonomy in multidatabases, or make the use of serializability inefficient like blocking long-running transactions.

FUTURE DIRECTIONS

A recent trend in transaction management focuses on adding transactional properties (e.g., isolation, atomicity) to business processes [3]. A business process (BP) is a set of tasks which are performed collaboratively to realize a business objective. Since BPs contain activities that access shared and persistent data resources, they have to be subject to transactional semantics. However, it is not adequate to treat an entire BP as a single "traditional" transaction mainly because BPs: (i) are of long duration and treating an entire process as a transaction would require locking resources for long periods of time, (ii) involve many independent database and application systems and enforcing transactional properties across the entire process would require expensive coordination among these systems, and (iii) have external effects and using conventional transactional rollback mechanisms is not feasible. These characteristics open new research issues to take the concept of correctness criterion and how it should be enforced beyond even the correctness criteria discussed here.

CROSS REFERENCE

Transaction Management
ACID properties

Concurrency Control
Two-Phase Locking
Recovery
Two-Phase Commit
Distributed, Parallel and Networked Databases

RECOMMENDED READING

[1] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] U. Dayal, M. Hsu, and R. Ladin. Business Process Coordination: State of the Art, Trends, and Open Issues. In *VLDB*, pages 3–13, 2001.

[4] W. Du and A. K. Elmagarmid. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *VLDB*, pages 347–355, 1989.

[5] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.

[6] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2), June 1983.

[7] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of Locks in a Large Shared Data Base. In *VLDB*, pages 428–451, 1975.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] H. F. Korth and G. D. Speegle. Formal Model of Correctness Without Serializability. In *SIGMOD Conference*, pages 379–386, 1988.

[10] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. Ensuring Consistency in Multidatabases by Preserving Two-Level Serializability. *ACM Trans. Database Syst.*, 23(2):199–230, 1998.

[11] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, 1979.

[12] C. H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[13] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Trans. Knowl. Data Eng.*, 7(6):997–1007, 1995.

[14] A. Sheth, Y. Leu, and A.K. Elmagarmid. Maintaining Consistency of Interdependent Data in Multidatabase Systems. Technical Report CSD-TR-91-016, Purdue University, http://www.cs.toronto.edu/ georgem/ws/ws.ps, March 1991.

[15] M. Yannakakis. Serializability by Locking. *J. ACM*, 31(2):227–244, 1984.