WIKIPEDIA

Serializability

In concurrency control of databases,[1][2] transaction processing (transaction management), and various transactional applications (e.g., transactional memory[3] and software transactional memory), both centralized and distributed, a transaction schedule is **serializable** if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems. *Strong strict two-phase locking* (SS2PL) is a popular serializability mechanism utilized in most of the database systems (in various variants) since their early days in the 1970s.

Serializability theory provides the formal framework to reason about and analyze serializability and its techniques. Though it is mathematical in nature, its fundamentals are informally (without mathematics notation) introduced below.

Contents

Correctness
- Serializability
- Relaxing serializability

View and conflict serializability

Enforcing conflict serializability
- Testing conflict serializability
- Common mechanism — SS2PL
- Other enforcing techniques
  - Optimistic versus pessimistic techniques
  - Serializable multi-version concurrency control

Distributed serializability
- Overview

See also

Notes

References

Correctness

Serializability

Serializability is used to keep the data in the data item in a consistent state. Serializability is a property of a transaction schedule (history). It relates to the *isolation* property of a database transaction.

> **Serializability** of a schedule means equivalence (in the outcome, the database state, data values) to a *serial schedule* (i.e., sequential with no transaction overlap in time) with the

same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems.

The rationale behind serializability is the following:
If each transaction is correct by itself, i.e., meets certain integrity conditions, then a schedule that comprises any *serial* execution of these transactions is correct (its transactions still meet their conditions): "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e, complete *isolation* between each other exists. Any order of the transactions is legitimate, if no dependencies among them exist, which is assumed (see comment below). As a result, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Schedules that are not serializable are likely to generate erroneous outcomes. Well known examples are with transactions that debit and credit accounts with money: If the related schedules are not serializable, then the total sum of money may not be preserved. Money could disappear, or be generated from nowhere. This and violations of possibly needed other invariant preservations are caused by one transaction writing, and "stepping on" and erasing what has been written by another transaction before it has become permanent in the database. It does not happen if serializability is maintained.

If any specific order between some transactions is requested by an application, then it is enforced independently of the underlying serializability mechanisms. These mechanisms are typically indifferent to any specific order, and generate some unpredictable partial order that is typically compatible with multiple serial orders of these transactions. This partial order results from the scheduling orders of concurrent transactions' data access operations, which depend on many factors.

A major characteristic of a database transaction is *atomicity*, which means that it either *commits*, i.e., all its operations' results take effect in the database, or *aborts* (rolled-back), all its operations' results do not have any effect on the database ("all or nothing" semantics of a transaction). In all real systems transactions can abort for many reasons, and serializability by itself is not sufficient for correctness. Schedules also need to possess the *recoverability* (from abort) property. **Recoverability** means that committed transactions have not read data written by aborted transactions (whose effects do not exist in the resulting database states). While serializability is currently compromised on purpose in many applications for better performance (only in cases when application's correctness is not harmed), compromising recoverability would quickly violate the database's integrity, as well as that of transactions' results external to the database. A schedule with the recoverability property (a *recoverable* schedule) "recovers" from aborts by itself, i.e., aborts do not harm the integrity of its committed transactions and resulting database. This is false without recoverability, where the likely integrity violations (resulting incorrect database data) need special, typically manual, corrective actions in the database.

Implementing recoverability in its general form may result in *cascading aborts*: Aborting one transaction may result in a need to abort a second transaction, and then a third, and so on. This results in a waste of already partially executed transactions, and may result also in a performance penalty. **Avoiding cascading aborts** (ACA, or Cascadelessness) is a special case of recoverability that exactly prevents such phenomena. Often in practice a special case of ACA is utilized: **Strictness**. Strictness allows efficient database recovery from failure.

Note that the *recoverability* property is needed even if no database failure occurs and no database *recovery* from failure is needed. It is, rather, needed to correctly automatically handle aborts, which may be unrelated to database failure and recovery from failure.

Relaxing serializability

In many applications, unlike with finances, absolute correctness is not needed. For example, when retrieving a list of products according to specification, in most cases it does not matter much if a product, whose data was updated a short time ago, does not appear in the list, even if it meets the specification. It will typically appear in such a list when tried again a short time later. Commercial databases provide concurrency control with a whole range of isolation levels which are in fact (controlled) serializability violations in order to achieve higher performance. Higher performance means better transaction execution rate and shorter average transaction response time (transaction duration). *Snapshot isolation* is an example of a popular, widely utilized efficient relaxed serializability method with many characteristics of full serializability, but still short of some, and unfit in many situations.

Another common reason nowadays for distributed serializability relaxation (see below) is the requirement of availability of internet products and services. This requirement is typically answered by large-scale data replication. The straightforward solution for synchronizing replicas' updates of the same database object is including all these updates in a single atomic distributed transaction. However, with many replicas such a transaction is very large, and may span enough of a number of several computers and networks that some of them are likely to be unavailable. Thus such a transaction is likely to end with abort and miss its purpose.[4] Consequently, Optimistic replication (Lazy replication) is often utilized (e.g., in many products and services by Google, Amazon, Yahoo, and the like), while serializability is relaxed and compromised for eventual consistency. Again, in this case, relaxation is done only for applications that are not expected to be harmed by this technique.

Classes of schedules defined by *relaxed serializability* properties either contain the serializability class, or are incomparable with it.

View and conflict serializability

Mechanisms that enforce serializability need to execute in real time, or almost in real time, while transactions are running at high rates. In order to meet this requirement, special cases of serializability, sufficient conditions for serializability which can be enforced effectively, are utilized.

Two major types of serializability exist: *view-serializability*, and *conflict-serializability*. View-serializability matches the general definition of serializability given above. Conflict-serializability is a broad special case, i.e., any schedule that is conflict-serializable is also view-serializable, but not necessarily the opposite. Conflict-serializability is widely utilized because it is easier to determine and covers a substantial portion of the view-serializable schedules. Determining view-serializability of a schedule is an NP-complete problem (a class of problems with only difficult-to-compute, excessively time-consuming known solutions).

> **View-serializability** of a schedule is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that respective transactions in the two schedules read and write the same data values ("view" the same data values).
>
> **Conflict-serializability** is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

Operations upon data are *read* or *write* (a write: either *insert* or *modify* or *delete*). Two operations are *conflicting* if they are of different transactions, upon the same datum (data item), and at least one of them is *write*. Each such pair of conflicting operations has a *conflict type*: It is either a *read-write*, or *write-read*, or a *write-write* conflict. The transaction of the second operation in the pair is said to be *in conflict* with the transaction of the first operation. A more general definition of conflicting operations (also for complex operations, which may each consist of several "simple" read/write operations) requires that they are noncommutative (changing their order also changes their combined result). Each such operation needs to be atomic by itself (using proper system support) in order to be considered an operation for a commutativity check. For example, read-read operations are commutative (unlike read-

write and the other possibilities) and thus read-read is not a conflict. Another more complex example: the operations *increment* and *decrement* of a *counter* are both *write* operations (both modify the counter), but do not need to be considered conflicting (write-write conflict type) since they are commutative (thus increment-decrement is not a conflict; e.g., already has been supported in the old IBM's IMS "fast path"). Only precedence (time order) in pairs of conflicting (non-commutative) operations is important when checking equivalence to a serial schedule, since different schedules consisting of the same transactions can be transformed from one to another by changing orders between different transactions' operations (different transactions' interleaving), and since changing orders of commutative operations (non-conflicting) does not change an overall operation sequence result, i.e., a schedule outcome (the outcome is preserved through order change between non-conflicting operations, but typically not when conflicting operations change order). This means that if a schedule can be transformed to any serial schedule without changing orders of conflicting operations (but changing orders of non-conflicting, while preserving operation order inside each transaction), then the outcome of both schedules is the same, and the schedule is conflict-serializable by definition.

Conflicts are the reason for blocking transactions and delays (non-materialized conflicts), or for aborting transactions due to serializability violation prevention. Both possibilities reduce performance. Thus reducing the number of conflicts, e.g., by commutativity (when possible), is a way to increase performance.

A transaction can issue/request a conflicting operation and be *in conflict* with another transaction while its conflicting operation is delayed and not executed (e.g., blocked by a lock). Only executed (*materialized*) conflicting operations are relevant to *conflict serializability* (see more below).

Enforcing conflict serializability

Testing conflict serializability

Schedule compliance with conflict serializability can be tested with the precedence graph (*serializability graph*, *serialization graph*, *conflict graph*) for committed transactions of the schedule. It is the directed graph representing precedence of transactions in the schedule, as reflected by precedence of conflicting operations in the transactions.

> In the **precedence graph** transactions are nodes and precedence relations are directed edges. There exists an edge from a first transaction to a second transaction, if the second transaction is *in conflict* with the first (see Conflict serializability above), and the conflict is **materialized** (i.e., if the requested conflicting operation is actually executed: in many cases a requested/issued conflicting operation by a transaction is delayed and even never executed, typically by a lock on the operation's object, held by another transaction, or when writing to a transaction's temporary private workspace and materializing, copying to the database itself, upon commit; as long as a requested/issued conflicting operation is not executed upon the database itself, the conflict is **non-materialized**; non-materialized conflicts are not represented by an edge in the precedence graph).
>
> **Comment:** In many text books only *committed transactions* are included in the precedence graph. Here all transactions are included for convenience in later discussions.

The following observation is a **key characterization of conflict serializability**:

> A schedule is *conflict-serializable* if and only if its precedence graph of *committed transactions* (when only *committed* transactions are considered) is *acyclic*. This means that a cycle consisting of committed transactions only is generated in the (general) precedence graph, if and only if conflict-serializability is violated.

Cycles of committed transactions can be prevented by aborting an *undecided* (neither committed, nor aborted) transaction on each cycle in the precedence graph of all the transactions, which can otherwise turn into a cycle of committed transactions (and a committed transaction cannot be aborted). One transaction aborted per cycle is both

required and sufficient in number to break and eliminate the cycle (more aborts are possible, and can happen under some mechanisms, but are unnecessary for serializability). The probability of cycle generation is typically low, but, nevertheless, such a situation is carefully handled, typically with a considerable amount of overhead, since correctness is involved. Transactions aborted due to serializability violation prevention are *restarted* and executed again immediately.

Serializability-enforcing mechanisms typically do not maintain a precedence graph as a data structure, but rather prevent or break cycles implicitly (e.g., SS2PL below).

Common mechanism — SS2PL

Strong strict two-phase locking (SS2PL) is a common mechanism utilized in database systems since their early days in the 1970s (the "SS" in the name SS2PL is newer, though) to enforce both conflict serializability and *strictness* (a special case of recoverability which allows effective database recovery from failure) of a schedule. Under this mechanism, each datum is locked by a transaction before its accessing it (in any read or write operation): the item is marked by and associated with a *lock* of a certain type depending on the operation being performed (and the specific transaction implementation; various models with different lock types exist; in some models, locks may change type during the transaction's life). As a result, access by another transaction may be blocked, typically upon a conflict (the lock delays or completely prevents the conflict from being materialized and be reflected in the precedence graph by blocking the conflicting operation), depending on lock type and the other transaction's access operation type. Employing an SS2PL mechanism means that all locks on data on behalf of a transaction are released only after the transaction has ended (either committed or aborted).

SS2PL is the name of the resulting schedule property as well, which is also called *rigorousness*. SS2PL is a special case (proper subset) of Two-phase locking (2PL)

Mutual blocking between transactions results in a *deadlock*, where execution of these transactions is stalled and no completion can be reached. Thus deadlocks need to be resolved to complete these transactions' execution and release related computing resources. A deadlock is a reflection of a potential cycle in the precedence graph that would occur without the blocking when conflicts are materialized. A deadlock is resolved by aborting a transaction involved with such a potential cycle and breaking the cycle. It is often detected using a *wait-for graph* (a graph of conflicts blocked by locks from being materialized; it can be also defined as the graph of non-materialized conflicts; conflicts not materialized are not reflected in the precedence graph and do not affect serializability), which indicates which transaction is "waiting for" the release of one of more locks by which other transaction or transactions, and a cycle in this graph means a deadlock. Aborting one transaction per cycle is sufficient to break the cycle. Transactions aborted due to deadlock resolution are *restarted* and executed again immediately.

Other enforcing techniques

Other known mechanisms include:

- Precedence graph (or Serializability graph, Conflict graph) cycle elimination
- Two-phase locking (2PL)
- Timestamp ordering (TO)
- Serializable snapshot isolation[5] (SerializableSI)

The above (conflict) serializability techniques in their general form do not provide recoverability. Special enhancements are needed for adding recoverability.

Optimistic versus pessimistic techniques

Concurrency control techniques are of three major types:

1. *Pessimistic*: In Pessimistic concurrency control, a transaction blocks the data access operations of other transactions upon conflicts, and conflicts are *non-materialized* until blocking is removed. This is done to ensure that operations that may violate serializability (and in practice also recoverability) do not occur.
2. *Optimistic*: In Optimistic concurrency control, the data access operations of other transactions are not blocked upon conflicts, and conflicts are immediately *materialized*. When the transaction reaches the *ready* state, i.e., its *running* state has been completed, possible serializability (and in practice also recoverability) violation by the transaction's operations (relative to other running transactions) is checked: if violation has occurred, the transaction is typically *aborted* (sometimes aborting *another* transaction to handle serializability violation is preferred). Otherwise, it is *committed*.
3. *Semi-optimistic*: Mechanisms that mix blocking in certain situations with not blocking in other situations and employ both materialized and non-materialized conflicts

The main differences between the technique types is the conflict types that are generated by them. A pessimistic method blocks a transaction operation upon conflict and generates a non-materialized conflict, while an optimistic method does not block and generates a materialized conflict. A semi-optimistic method generates both conflict types. Both conflict types are generated by the chronological orders in which transaction operations are invoked, independently of the type of conflict. A cycle of committed transactions (with materialized conflicts) in the *precedence graph* (conflict graph) represents a serializability violation, and should be avoided for maintaining serializability. A cycle of (non-materialized) conflicts in the *wait-for graph* represents a deadlock situation, which should be resolved by breaking the cycle. Both cycle types result from conflicts and should be broken. Under any technique type, conflicts should be detected and considered, with similar overhead for both materialized and non-materialized conflicts (typically by using mechanisms like locking, while either blocking for locks or not blocking but recording conflict for materialized conflicts). In a blocking method, typically a context switching occurs upon conflict, with (additional) incurred overhead. Otherwise, blocked transactions' related computing resources remain idle, unutilized, which may be a worse alternative. When conflicts do not occur frequently, optimistic methods typically have an advantage. With different transaction loads (mixes of transaction types,) one technique type (i.e., either optimistic or pessimistic) may provide better performance than the other.

Unless schedule classes are *inherently blocking* (i.e., they cannot be implemented without data-access operations blocking; e.g., 2PL, SS2PL and SCO above; see chart), they can also be implemented using optimistic techniques (e.g., Serializability, Recoverability).

Serializable multi-version concurrency control

> See also Multiversion concurrency control (partial coverage) and Serializable Snapshot Isolation in Snapshot isolation

Multi-version concurrency control (MVCC) is a common way today to increase concurrency and performance by generating a new version of a database object each time the object is written and allowing transactions' read operations of several last relevant versions (of each object), depending on scheduling method. MVCC can be combined with all the serializability techniques listed above (except SerializableSI, which is originally MVCC-based). It is utilized in most general-purpose DBMS products.

MVCC is especially popular nowadays through the *relaxed serializability* (see above) method *Snapshot isolation* (SI,) which provides better performance than most known serializability mechanisms (at the cost of possible serializability violation in certain cases). SerializableSI, which is an efficient enhancement of SI to make it serializable, is intended to provide an efficient serializable solution. SerializableSI has been analyzed[5][6] via a general theory of MVCC

Distributed serializability

Overview

Distributed serializability is the serializability of a schedule of a transactional distributed system (e.g., a distributed database system). Such a system is characterized by *distributed transactions* (also called *global transactions*), i.e., transactions that span computer processes (a process abstraction in a general sense, depending on computing environment; e.g., operating system's thread) and possibly network nodes. A distributed transaction comprises more than one of several *local sub-transactions* that each has states as described above for a database transaction. A local sub-transaction comprises a single process, or more processes that typically fail together (e.g., in a single processor core). Distributed transactions imply a need for an atomic commit protocol to reach consensus among its local sub-transactions on whether to commit or abort. Such protocols can vary from a simple (one-phase) handshake among processes that fail together to more sophisticated protocols, like two-phase commit, to handle more complicated cases of failure (e.g., process, node, communication, etc. failure). Distributed serializability is a major goal of distributed concurrency control for correctness. With the proliferation of the Internet, cloud computing, grid computing, and small, portable, powerful computing devices (e.g., smartphones,) the need for effective distributed serializability techniques to ensure correctness in and among distributed applications seems to increase.

Distributed serializability is achieved by implementing distributed versions of the known centralized techniques.[1][2] Typically, all such distributed versions require utilizing conflict information (of either materialized or non-materialized conflicts, or, equivalently, transaction precedence or blocking information; conflict serializability is usually utilized) that is not generated locally, but rather in different processes, and remote locations. Thus information distribution is needed (e.g., precedence relations, lock information, timestamps, or tickets). When the distributed system is of a relatively small scale and message delays across the system are small, the centralized concurrency control methods can be used unchanged while certain processes or nodes in the system manage the related algorithms. However, in a large-scale system (e.g., *grid* and *cloud*), due to the distribution of such information, a substantial performance penalty is typically incurred, even when distributed versions of the methods (vs. the centralized ones) are used, primarily due to computer and communication latency. Also, when such information is distributed, related techniques typically do not scale well. A well-known example with respect to scalability problems is a distributed lock manager, which distributes lock (non-materialized conflict) information across the distributed system to implement locking techniques.

See also

- Strong strict two-phase locking (SS2PL or Rigorousness).
- Making snapshot isolation serializable[5] in Snapshot isolation.
- Global serializability, where the *Global serializability problem* and its proposed solutions are described.
- Linearizability, a more general concept in concurrent computing

Notes

1. Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman (1987): *Concurrency Control and Recovery in Database Systems* (http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx) (free PDF download), Addison Wesley Publishing Company, ISBN 0-201-10715-5
2. Gerhard Weikum, Gottfried Vossen (2001): *Transactional Information Systems* (http://www.elsevier.com/wps/find/bookdescription.cws_home/677937/description#description), Elsevier, ISBN 1-55860-508-8
3. Maurice Herlihy and J. Eliot B. Moss. *Transactional memory: architectural support for lock-free data structures.* Proceedings of the 20th annual international symposium on Computer architecture (ISCA '93). Volume 21, Issue 2, May 1993.
4. Gray, J.; Helland, P.; O'Neil, P.; Shasha, D. (1996). *The dangers of replication and a solution* (ftp://ftp.research.microsoft.com/pub/tr/tr-96-17.pdf) (PDF). Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. pp. 173–182. doi:10.1145/233269.233330 (https://doi.org/10.1145%2F233269.233330).
5. Michael J. Cahill, Uwe Röhm, Alan D. Fekete (2008): "Serializable isolation for snapshot databases" (http://portal.acm.org/citation.cfm?id=1376690), *Proceedings of the 2008 ACM SIGMOD international conference on*

Management of data, pp. 729-738, Vancouver, Canada, June 2008, ISBN 978-1-60558-102-6 (SIGMOD 2008 best paper award)

6. Alan Fekete (2009), "Snapshot Isolation and Serializable Execution" (http://www.it.usyd.edu.au/~fekete/teaching/serializableSI-Fekete.pdf), Presentation, Page 4, 2009, The university of Sydney (Australia). Retrieved 16 September 2009

References

- Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman (1987): *Concurrency Control and Recovery in Database Systems* (http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx), Addison Wesley Publishing Company, ISBN 0-201-10715-5
- Gerhard Weikum, Gottfried Vossen (2001): *Transactional Information Systems* (http://www.elsevier.com/wps/find/bookdescription.cws_home/677937/description#description), Elsevier, ISBN 1-55860-508-8