Transaction properties(ACID properties)

In computer science, **ACID** (*Atomicity*, *Consistency*, *Isolation*, *Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context ofdatabases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Jim Gray defined these properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically.

In 1983, Andreas Reuter and Theo Härder coined the acronym *ACID* to describe them.

The characteristics of these four properties as defined by Reuter and Härder:

Atomicity

Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

Consistency

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.

Isolation

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method (i.e. if it uses strict - as opposed to relaxed - serializability), the effects of an incomplete transaction might not even be visible to another transaction.

Durability

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL

statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

Concurrency control

In information technology and computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, concurrency control**ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.**

Computer systems, both software and hardware, consist of modules, or components. Each component is designed to operate correctly, i.e., to obey or to meet certain consistency rules. When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. The general area of concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints which typically result in some performance reduction. Operation consistency and correctness should be achieved with as good as possible efficiency, without reducing performance below reasonable levels. Concurrency control can require significant additional complexity and overhead in a concurrent algorithmcompared to the simpler sequential algorithm.

Why is concurrency control needed?[edit]

If transactions are executed *serially*, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur, such as:

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.

3. The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

Most high-performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their databases consistent.

Concurrency control mechanisms[edit]

***Categories**[edit]*

The main categories of concurrency control mechanisms are:

- **Optimistic** - Delay the checking of whether a transaction meets the isolation and other integrity rules (e.g., serializability and recoverability) until its end, without blocking any of its (read, write) operations ("...and be optimistic about the rules being met..."), and then abort a transaction to prevent the violation, if the desired rules are to be violated upon its commit. An aborted transaction is immediately restarted and re-executed, which incurs an obvious overhead (versus executing it to the end only once). If not too many transactions are aborted, then being optimistic is usually a good strategy.
- **Pessimistic** - Block an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction.
- **Semi-optimistic** - Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking (if needed) to transaction's end, as done with optimistic.

Different categories provide different performance, i.e., different average transaction completion rates (*throughput*), depending on transaction types mix, computing level of parallelism, and other factors. If selection and knowledge about trade-offs are available, then category and method should be chosen to provide the highest performance.

The mutual blocking between two transactions (where each one blocks the other) or more results in a deadlock, where the transactions involved are stalled and cannot reach completion. Most non-optimistic mechanisms (with blocking) are prone to deadlocks which are resolved by an intentional abort of a stalled transaction (which releases the other transactions in that deadlock), and its immediate restart and re-execution. The likelihood of a deadlock is typically low.

Blocking, deadlocks, and aborts all result in performance reduction, and hence the trade-offs between the categories.

***Methods**[edit]*

Many methods for concurrency control exist. Most of them can be implemented within either main category above. The major methods,[1] which have each many variants, and in some cases may overlap or be combined, are:

1. Locking (e.g., **Two-phase locking** - 2PL) - Controlling access to data by locks assigned to the data. Access of a transaction to a data item (database object) locked by another transaction may be blocked (depending on lock type and access operation type) until lock release.
2. **Serialization graph checking** (also called Serializability, or Conflict, or Precedence graph checking) - Checking for cycles in the schedule's graph and breaking them by aborts.
3. **Timestamp ordering** (TO) - Assigning timestamps to transactions, and controlling or checking access to data by timestamp order.
4. **Commitment ordering** (or Commit ordering; CO) - Controlling or checking transactions' chronological order of commit events to be compatible with their respectiveprecedence order.

Other major concurrency control types that are utilized in conjunction with the methods above include:

- **Multiversion concurrency control** (MVCC) - Increasing 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.
- **Index concurrency control** - Synchronizing access operations to indexes, rather than to user data. Specialized methods provide substantial performance gains.
- **Private workspace model** (**Deferred update**) - Each transaction maintains a private workspace for its accessed data, and its changed data become visible outside the transaction only upon its commit (e.g., Weikum and Vossen 2001). This model provides a different concurrency control behavior with benefits in many cases.

The most common mechanism type in database systems since their early days in the 1970s has been *Strong strict Two-phase locking* (SS2PL; also called *Rigorous scheduling* or*Rigorous 2PL*) which is a special case (variant) of both Two-phase locking (2PL) and Commitment ordering (CO). It is pessimistic. In spite of its long name (for historical reasons) the idea of the **SS2PL** mechanism is simple: "Release all locks applied by a transaction only after the transaction has ended." SS2PL (or

Rigorousness) is also the name of the set of all schedules that can be generated by this mechanism, i.e., these are SS2PL (or Rigorous) schedules, have the SS2PL (or Rigorousness) property.

Major goals of concurrency control mechanisms[edit]

Concurrency control mechanisms firstly need to operate correctly, i.e., to maintain each transaction's integrity rules (as related to concurrency; application-specific integrity rule are out of the scope here) while transactions are running concurrently, and thus the integrity of the entire transactional system. Correctness needs to be achieved with as good performance as possible. In addition, increasingly a need exists to operate effectively while transactions are distributed over processes, computers, and computer networks. Other subjects that may affect concurrency control are recovery and replication.

Correctness

For correctness, a common major goal of most concurrency control mechanisms is generating schedules with the *Serializability* property. Without serializability undesirable phenomena may occur, e.g., money may disappear from accounts, or be generated from nowhere. **Serializability** of a schedule means equivalence (in the resulting database values) to some *serial* schedule with the same transactions (i.e., in which transactions are sequential with no overlap in time, and thus completely isolated from each other: No concurrent access by any two transactions to the same data is possible). Serializability is considered the highest level of isolation among database transactions, and the major correctness criterion for concurrent transactions. In some cases compromised, relaxed forms of serializability are allowed for better performance (e.g., the popular *Snapshot isolation* mechanism) or to meet availability requirements in highly distributed systems (see *Eventual consistency*), but only if application's correctness is not violated by the relaxation (e.g., no relaxation is allowed for money transactions, since by relaxation money can disappear, or appear from nowhere).

Almost all implemented concurrency control mechanisms achieve serializability by providing *Conflict serializablity*, a broad special case of serializability (i.e., it covers, enables most serializable schedules, and does not impose significant additional delay-causing constraints) which can be implemented efficiently.

Recoverability

> **Comment:** While in the general area of systems the term "recoverability" may refer to the ability of a system to recover from failure or from an incorrect/forbidden state, within concurrency control of database systems this term has received a specific meaning.
>
> Concurrency control typically also ensures the *Recoverability* property of schedules for maintaining correctness in cases of aborted transactions (which can always happen for many reasons). **Recoverability** (from abort) means that no committed transaction in a schedule has

read data written by an aborted transaction. Such data disappear from the database (upon the abort) and are parts of an incorrect database state. Reading such data violates the consistency rule of ACID. Unlike Serializability, Recoverability cannot be compromised, relaxed at any case, since any relaxation results in quick database integrity violation upon aborts. The major methods listed above provide serializability mechanisms. None of them in its general form automatically provides recoverability, and special considerations and mechanism enhancements are needed to support recoverability. A commonly utilized special case of recoverability is *Strictness*, which allows efficient database recovery from failure (but excludes optimistic implementations; e.g.,Strict CO (SCO) cannot have an optimistic implementation, but has semi-optimistic ones).

Comment: 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 transaction aborts, which may be unrelated to database failure and recovery from it.

Distribution

With the fast technological development of computing the difference between local and distributed computing over low latency networks or buses is blurring. Thus the quite effective utilization of local techniques in such distributed environments is common, e.g., in computer clusters and multi-core processors. However the local techniques have their limitations and use multi-processes (or threads) supported by multi-processors (or multi-cores) to scale. This often turns transactions into distributed ones, if they themselves need to span multi-processes. In these cases most local concurrency control techniques do not scale well.

> As database systems have become distributed, or started to cooperate in distributed environments (e.g., Federated databases in the early 1990s, and nowadays Grid computing,Cloud computing, and networks with smartphones), some transactions have become distributed. A distributed transaction means that the transaction spans processes, and may span computers and geographical sites. This generates a need in effective distributed concurrency control mechanisms. Achieving the Serializability property of a distributed system's schedule (see *Distributed serializability* and *Global serializability* (*Modular serializability*)) effectively poses special challenges typically not met by most of the regular serializability mechanisms, originally designed to operate locally. This is especially due to a need in costly distribution of concurrency control information amid communication and computer latency. The only known general effective technique for distribution is Commitment ordering, which was disclosed publicly in 1991 (after being patented). **Commitment ordering** (Commit ordering, CO; Raz 1992) means that transactions' chronological order of commit events is kept compatible with their respective precedence order. CO does not require the distribution of concurrency control information and provides a general effective solution (reliable, high-performance, and scalable) for both distributed and global serializability, also in a heterogeneous

environment with database systems (or other transactional objects) with different (any) concurrency control mechanisms.[1] CO is indifferent to which mechanism is utilized, since it does not interfere with any transaction operation scheduling (which most mechanisms control), and only determines the order of commit events. Thus, CO enables the efficient distribution of all other mechanisms, and also the distribution of a mix of different (any) local mechanisms, for achieving distributed and global serializability. The existence of such a solution has been considered "unlikely" until 1991, and by many experts also later, due to misunderstanding of the CO solution(see Quotations in *Global serializability*). An important side-benefit of CO is automatic distributed deadlock resolution. Contrary to CO, virtually all other techniques (when not combined with CO) are prone to distributed deadlocks (also called global deadlocks) which need special handling. CO is also the name of the resulting schedule property: A schedule has the CO property if the chronological order of its transactions' commit events is compatible with the respective transactions' precedence (partial) order.

SS2PL mentioned above is a variant (special case) of CO and thus also effective to achieve distributed and global serializability. It also provides automatic distributed deadlock resolution (a fact overlooked in the research literature even after CO's publication), as well as Strictness and thus Recoverability. Possessing these desired properties together with known efficient locking based implementations explains SS2PL's popularity. SS2PL has been utilized to efficiently achieve Distributed and Global serializability since the 1980, and has become the de facto standard for it. However, SS2PL is blocking and constraining (pessimistic), and with the proliferation of distribution and utilization of systems different from traditional database systems (e.g., as in Cloud computing), less constraining types of CO (e.g., Optimistic CO) may be needed for better performance.

Distributed recoverability

Unlike Serializability, *Distributed recoverability* and *Distributed strictness* can be achieved efficiently in a straightforward way, similarly to the way Distributed CO is achieved: In each database system they have to be applied locally, and employ a vote ordering strategy for the Two-phase commit protocol (2PC; Raz 1992, page 307).

As has been mentioned above, Distributed SS2PL, including Distributed strictness (recoverability) and Distributed commitment ordering (serializability), automatically employs the needed vote ordering strategy, and is achieved (globally) when employed locally in each (local) database system (as has been known and utilized for many years; as a matter of fact locality is defined by the boundary of a 2PC participant (Raz 1992)).

Other major subjects of attention

The design of concurrency control mechanisms is often influenced by the following subjects:

Recovery

All systems are prone to failures, and handling *recovery* from failure is a must. The properties of the generated schedules, which are dictated by the concurrency control mechanism, may have an impact on the effectiveness and efficiency of recovery. For example, the Strictness property (mentioned in the section Recoverability above) is often desirable for an efficient recovery.

CONCURRENCY CONTROL FOR SCHEDULER

Formal description

The following is an example of a schedule:

$$D = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ W(X) & & \\ Com. & & \\ & R(Y) & \\ & W(Y) & \\ & Com. & \\ & & R(Z) \\ & & W(Z) \\ & & Com. \end{bmatrix}$$

In this example, the horizontal axis represents the different transactions in the schedule D. The vertical axis represents time order of operations. Schedule D consists of three transactions T1, T2, T3. The schedule describes the actions of the transactions as seen by the DBMS. First T1 Reads and Writes to object X, and then Commits. Then T2 Reads and Writes to object Y and Commits, and finally T3 Reads and Writes to object Z and Commits. This is an example of a *serial* schedule, i.e., sequential with no overlap in time, because the actions of in all three transactions are sequential, and the transactions are not interleaved in time.

Representing the schedule D above by a table (rather than a list) is just for the convenience of identifying each transaction's operations in a glance. This notation is used throughout the article below. A more common way in the technical literature for representing such schedule is by a list:

D = R1(X) W1(X) Com1 R2(Y) W2(Y) Com2 R3(Z) W3(Z) Com3

Usually, for the purpose of reasoning about concurrency control in databases, an operation is modeled as *atomic*, occurring at a point in time, without duration. When this is not

satisfactory start and end time-points and possibly other point events are specified (rarely). Real executed operations always have some duration and specified respective times of occurrence of events within them (e.g., "exact" times of beginning and completion), but for concurrency control reasoning usually only the precedence in time of the whole operations (without looking into the quite complex details of each operation) matters, i.e., which operation is before, or after another operation. Furthermore, in many cases the before/after relationships between two specific operations do not matter and should not be specified, while being specified for other pairs of operations.

In general operations of transactions in a schedule can interleave (i.e., transactions can be executed concurrently), while time orders between operations in each transaction remain unchanged as implied by the transaction's program. Since not always time orders between all operations of all transactions matter and need to be specified, a schedule is, in general, a *partial order* between operations rather than a *total order* (where order for each pair is determined, as in a list of operations). Also in the general case each transaction may consist of several processes, and itself be properly represented by a partial order of operations, rather than a total order. Thus in general a schedule is a partial order of operations, containing (embedding) the partial orders of all its transactions.

Time-order between two operations can be represented by an *ordered pair* of these operations (e.g., the existence of a pair (OP1,OP2) means that OP1 is always before OP2), and a schedule in the general case is a set of such ordered pairs. Such a set, a schedule, is a partial order which can be represented by an *acyclic directed graph* (or *directed acyclic graph*, DAG) with operations as nodes and time-order as a directed edge (no cycles are allowed since a cycle means that a first (any) operation on a cycle can be both before and after (any) another second operation on the cycle, which contradicts our perception of Time). In many cases a graphical representation of such graph is used to demonstrate a schedule.

Comment: Since a list of operations (and the table notation used in this article) always represents a total order between operations, schedules that are not a total order cannot be represented by a list (but always can be represented by a DAG).

Types of schedule

Serial

The transactions are executed non-interleaved (see example above) i.e., a serial schedule is one in which no transaction starts until a running transaction has ended.

Serializable

A schedule that is equivalent (in its outcome) to a serial schedule has the serializability property.

In schedule E, the order in which the actions of the transactions are executed is not the same as in D, but in the end, E gives the same result as D.

$$E = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ & R(Y) & \\ & & R(Z) \\ W(X) & & \\ & W(Y) & \\ & & W(Z) \\ Com. & Com. & Com. \end{bmatrix}$$

Conflicting actions

Two actions are said to be in conflict (conflicting pair) if:

1. The actions belong to different transactions.
2. At least one of the actions is a write operation.
3. The actions access the same object (read or write).

The following set of actions is conflicting:

- R1(X), W2(X), W3(X) (3 conflicting pairs)

While the following sets of actions are not:

- R1(X), R2(X), R3(X)
- R1(X), W2(Y), R3(X)

Conflict equivalence

The schedules S1 and S2 are said to be conflict-equivalent if following two conditions are satisfied:

1. Both schedules S1 and S2 involve the same set of transactions (including ordering of actions within each transaction).
2. Both schedules have same set of conflicting operations.

Conflict-serializable

A schedule is said to be conflict-serializable when the schedule is conflict-equivalent to one or more serial schedules.

Another definition for conflict-serializability is that a schedule is conflict-serializable if and only if its precedence graph/serializability graph, when only committed transactions

are considered, is acyclic (if the graph is defined to include also uncommitted transactions, then cycles involving uncommitted transactions may occur without conflict serializability violation).

$$G = \begin{bmatrix} T1 & T2 \\ R(A) & \\ & R(A) \\ W(B) & \\ Com. & \\ & W(A) \\ & Com. \\ & \end{bmatrix}$$

Which is conflict-equivalent to the serial schedule <T1,T2>, but not <T2,T1>.

Commitment ordering

A schedule is said to be commitment-ordered (commit-ordered), or commitment-order-serializable, if it obeys the Commitment ordering (CO; also commit-ordering or commit-order-serializability) schedule property. This means that the order in time of transactions' commitment events is compatible with the precedence (partial) order of the respective transactions, as induced by their schedule's acyclic precedence graph (serializability graph, conflict graph). This implies that it is also conflict-serializable. The CO property is especially effective for achieving Global serializability in distributed systems.

.

View equivalence

Two schedules S1 and S2 are said to be view-equivalent when the following conditions are satisfied:

1. If the transaction T_i in S1 reads an initial value for object X, so does the transaction T_i in S2.
2. If the transaction T_i in S1 reads the value written by transaction T_j in S1 for object X, so does the transaction T_i in S2.
3. If the transaction T_i in S1 is the final transaction to write the value for an object X, so is the transaction T_i in S2.

View-serializable

A schedule is said to be view-serializable if it is view-equivalent to some serial schedule. Note that by definition, all conflict-serializable schedules are view-serializable.

$$G = \begin{bmatrix} T1 & T2 \\ R(A) & \\ & R(A) \\ W(B) & \end{bmatrix}$$

Notice that the above example (which is the same as the example in the discussion of conflict-serializable) is both view-serializable and conflict-serializable at the same time.) There are however view-serializable schedules that are not conflict-serializable: those schedules with a transaction performing a blind write:

$$H = \begin{bmatrix} T1 & T2 & T3 \\ R(A) & & \\ & W(A) & \\ & Com. & \\ W(A) & & \\ Com. & & \\ & & W(A) \\ & & Com. \\ & & \end{bmatrix}$$

The above example is not conflict-serializable, but it is view-serializable since it has a view-equivalent serial schedule <T1, T2, T3>.

Since determining whether a schedule is view-serializable is NP-complete, view-serializability has little practical interest.

Recoverable

Transactions commit only after all transactions whose changes they read, commit.

$$F = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Com. & \\ & Com. \\ & \end{bmatrix} \quad F2 = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Abort & \\ & Abort \\ & \end{bmatrix}$$

These schedules are recoverable. F is recoverable because T1 commits before T2, that makes the value read by T2 correct. Then T2 can commit itself. In F2, if T1 aborted, T2 has to abort because the value of A it read is incorrect. In both cases, the database is left in a consistent state.

Unrecoverable

If a transaction T1 aborts, and a transaction T2 commits, but T2 relied on T1, we have an unrecoverable schedule.

$$G = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ & Com. \\ Abort & \\ & \end{bmatrix}$$

In this example, G is unrecoverable, because T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is unrecoverable.

Avoids cascading aborts (rollbacks)

Also named cascadeless. A single transaction abort leads to a series of transaction rollback. Strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

The following examples are the same as the one from the discussion on recoverable:

$$F = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Com. & \\ & Com. \\ & \end{bmatrix} \quad F2 = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Abort & \\ & Abort \\ & \end{bmatrix}$$

In this example, although F2 is recoverable, it does not avoid cascading aborts. It can be seen that if T1 aborts, T2 will have to be aborted too in order to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

The following is a recoverable schedule which avoids cascading abort. Note, however, that the update of A by T1 is always lost (since T1 is aborted).

$$F3 = \begin{bmatrix} T1 & T2 \\ & R(A) \\ R(A) & \\ W(A) & \\ & W(A) \\ Abort & \\ & Commit \\ & \end{bmatrix}$$

Cascading aborts avoidance is sufficient but not necessary for a schedule to be recoverable.

Strict

A schedule is strict - has the strictness property - if for any two transactions T1, T2, if a write operation of T1 precedes a *conflicting* operation of T2 (either read or write), then the commit event of T1 also precedes that conflicting operation of T2.

Any strict schedule is cascadeless, but not the converse. Strictness allows efficient recovery of databases from failure.