ACID

Connected to:

Database Andreas Reuter Theo Härder

From Wikipedia, the free encyclopedia

In computer science, **ACID** (*Atomicity*, *Consistency*, *Isolation*, *Durability*) is a set of properties of database transactions. In the context of databases, 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.[1][2][3]

In 1983, Andreas Reuter and Theo Härder coined the acronym *ACID* to describe them.[4]

Characteristics

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

Atomicity

Main article: Atomicity (database systems)

Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, then 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

Main article: Consistency (database systems)

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

Main article: Isolation (database systems)

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 the 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

Main article: Durability (database systems)

The durability property ensures 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.

Examples

The following examples further illustrate the ACID properties. In these examples, the database table has two columns, A and B. An integrity constraint requires that the value in A and the value in B must sum to 100. The following SQL code creates a table as described above:

```
CREATE TABLE acidtest (A INTEGER, B INTEGER, CHECK (A + B = 100));
```

Atomicity failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

In database systems, atomicity (or atomicness; from Greek a-tomos, undividable) is one of the ACID transaction properties. In an atomic transaction, a series of database operations either all occur, or nothing occurs. The series of operations cannot be divided apart and executed partially from each other, which makes the series of operations "indivisible", hence the name. A guarantee of atomicity prevents updates to the database occurring only partially, which can cause greater problems than rejecting the whole series outright. In other words, atomicity means indivisibility and irreducibility.

Consistency failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Consistency is a very general term, which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that A + B = 100. Also, it may be inferred that both A and B must be integers. A valid range for A and B may also be inferred. All validation rules must be checked to ensure consistency. Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that A + B = 100 before the transaction begins. If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that A + B = 90, which is inconsistent with the rules of the database. The entire transaction must be cancelled and the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed.

Isolation failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation.

Consider two transactions. T_1 transfers 10 from A to B. T_2 transfers 10 from B to A. Combined, there are four actions:

- T_1 subtracts 10 from A.
- T_1 adds 10 to B.
- T_2 subtracts 10 from B.
- T_2 adds 10 to A.

If these operations are performed in order, isolation is maintained, although T_2 must wait. Consider what happens if T_1 fails halfway through. The database eliminates T_1's effects, and T_2 sees only valid data.

By interleaving the transactions, the actual order of actions might be:

- T_1 subtracts 10 from A.
- T_2 subtracts 10 from B.
- T_2 adds 10 to A.
- T_1 adds 10 to B.

Again, consider what happens if T_1 fails halfway through. By the time T_1 fails, T_2 has already modified A; it cannot be restored to the value it had before T_1 without leaving an invalid database. This is known as a write-write failure,[*citation needed*] because two transactions attempted to write to the same data field. In a typical system, the problem would be resolved by reverting to the last known good state, canceling the failed transaction T_1, and restarting the interrupted transaction T_2 from the good state.

Durability failure

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Consider a transaction that transfers 10 from A to B. First it removes 10 from A, then it adds 10 to B. At this point, the user is told the transaction was a success, however the changes are still queued in the disk buffer waiting to be committed to disk. Power fails and the changes are lost. The user assumes (understandably) that the changes have been persisted.

Implementation

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons. For instance, the system may have no room left on its disk drives, or it may have used up its allocated CPU time. There are two popular families of techniques: write-ahead logging and shadow paging. In both cases, locks must be acquired on all information to be updated, and depending on the level of isolation, possibly on all data that be read as well. In write ahead logging, atomicity is guaranteed by copying the original (unchanged) data to a log before changing the database.[*dubious* – *discuss*] That allows the database to return to a consistent state in the event of a crash. In shadowing, updates are applied to a partial copy of the database, and the new copy is activated when the transaction commits.

Locking vs multiversioning

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Many databases rely upon locking to provide ACID capabilities. Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify it until the first transaction succeeds or fails. The lock must always be acquired before processing data, including data that is read but not modified. Non-trivial transactions typically require a large number of locks, resulting in substantial overhead as well as blocking other transactions. For example, if user A is running a transaction that has to read a row of data that user B wants to modify, user B must wait until user A's transaction completes. Two phase locking is often applied to guarantee full isolation.

An alternative to locking is multiversion concurrency control, in which the database provides each reading transaction the prior, unmodified version of data that is being modified by another active transaction. This allows readers to operate without acquiring locks, i.e. writing transactions do not block reading transactions, and readers do not block writers. Going back to the example, when user A's transaction requests data that user B is modifying, the database provides A with the version of that data that existed when user B started his transaction. User A gets a consistent view of the database even if other users are changing data. One implementation, namely snapshot isolation, relaxes the isolation property.

Distributed transactions

This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2015) (Learn how and when to remove this template message)

Main article: Distributed transaction

Guaranteeing ACID properties in a distributed transaction across a distributed database, where no single node is responsible for all data affecting a transaction, presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes because of a failure on another node. The two-phase commit protocol (not to be confused with two-phase locking) provides atomicity for distributed transactions to ensure that each participant in the transaction agrees on whether the transaction should be committed or not.[*citation needed*] Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants) and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transaction.

See also

- Basically Available, Soft state, Eventual consistency (BASE)
- CAP theorem
- Concurrency control
- Java Transaction API
- Open Systems Interconnection
- Transactional NTFS
- Two-phase commit protocol
- CRUD

References

1. ^ *"Gray to be Honored With A.M. Turing Award This Spring"*. *Microsoft PressPass*. *Archived from the original on February 6, 2009*. *Retrieved March 27, 2015*.
2. ^ *Gray, Jim (September 1981)*. *"The Transaction Concept: Virtues and Limitations" (PDF)*. *Proceedings of the 7th International Conference on Very Large Databases*. *Cupertino, CA: Tandem Computers*. *pp. 144–154*. *Retrieved March 27, 2015*.
3. ^ Gray, Jim & Andreas Reuter. *Distributed Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993; ISBN 1-55860-190-2.
4. ^ *Haerder, T.; Reuter, A. (1983)*. "*Principles of transaction-oriented database recovery*". *ACM Computing Surveys*. ***15*** *(4): 287*. *doi:10.1145/289.291*. These four properties, atomicity, consistency, isolation, and durability (ACID), describe the major highlights of the transaction paradigm, which has influenced many aspects of development in database systems.

[hide]

- **v**
- **t**
- **e**

Database management systems

Types

- Object-oriented
  - comparison
- Relational
  - comparison
- Document-oriented
- Graph
- NoSQL
- NewSQL