ACID

In computer science, **ACID** (atomicity, consistency, isolation, durability) is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps. In the context of databases, a sequence of database operations that satisfies the ACID properties (which can be perceived as 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.

In 1983,[1] Andreas Reuter and Theo Härder coined the acronym *ACID*, building on earlier work by Jim Gray[2] who named atomicity, consistency, and durability, but not isolation, when characterizing the transaction concept. These four properties are the major guarantees of the transaction paradigm, which has influenced many aspects of development in database systems.

According to Gray and Reuter, the IBM Information Management System supported ACID transactions as early as 1973 (although the acronym was created later).[3]

BASE stands for basically available, soft state, and eventually consistent: the acronym highlights that BASE is opposite of ACID, like their chemical equivalents.[4] ACID databases prioritize consistency over **availability** — the whole transaction fails if an error occurs in any step within the transaction; in contrast, BASE databases prioritize **availability** over **consistency**: instead of failing the transaction, users can access inconsistent data **temporarily**: data consistency is achieved, but **not immediately**.[5]

Characteristics

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

Atomicity

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes.[6] A guarantee of atomicity prevents updates to the database from occurring only partially, which can cause greater problems than rejecting the whole series outright. As a consequence, the transaction cannot be observed to be in progress by another database client. At one moment in time, it has not yet happened, and at the next, it has already occurred in whole (or nothing happened if the transaction was cancelled in progress).

Consistency

Consistency ensures that a transaction can only bring the database from one consistent state to another, preserving database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction. An example of a database invariant is referential integrity, which guarantees the primary key–foreign key relationship.[7]

Isolation

Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Isolation is the main goal of concurrency control; depending on the isolation level used, the effects of an incomplete transaction might not be visible to other transactions.[8]

Durability

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.[9]

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

Atomicity is the guarantee that series of database operations in an atomic transaction will either all occur (a successful operation), or none will occur (an unsuccessful operation). The series of operations cannot be separated with only some of them being executed, which makes the series of operations "indivisible". A guarantee of atomicity prevents updates to the database from occurring only partially, which can cause greater problems than rejecting the whole series outright. In other words, atomicity means indivisibility and irreducibility.[10] Alternatively, we may say that a logical transaction may be composed of several physical transactions. Unless and until all component physical transactions are executed, the logical transaction will not have occurred.

An example of an atomic transaction is a monetary transfer from bank account A to account B. It consists of two operations, withdrawing the money from account A and depositing it to account B. We would not want to see the amount removed from account A before we are sure it has also been

transferred into account B. Performing these operations in an atomic transaction ensures that the database remains in a consistent state, that is, money is neither debited nor credited if either of those two operations fails.[11]

Consistency failure

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 $A + B = 100$. 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 canceled 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. Similar issues may arise with other constraints. We may have required the data types of both A and B to be integers. If we were then to enter, say, the value 13.5 for A, the transaction will be canceled, or the system may give rise to an alert in the form of a trigger (if/when the trigger has been written to this effect). Another example would be integrity constraints, which would not allow us to delete a row in one table whose primary key is referred to by at least one foreign key in other tables.

Isolation failure

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 20 from B to A.

Combined, there are four actions:

1. T_1 subtracts 10 from A.
2. T_1 adds 10 to B.
3. T_2 subtracts 20 from B.
4. T_2 adds 20 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:

1. T_1 subtracts 10 from A.
2. T_2 subtracts 20 from B.

3. T_2 adds 20 to A.
4. T_1 adds 10 to B.

Again, consider what happens if T_1 fails while modifying B in Step 4. 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 contention,[12] 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

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, but the user assumes (understandably) that the changes persist.

Implementation

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 may be read as well. In write ahead logging, durability is guaranteed by writing the prospective change to a persistent log before changing the database. 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

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

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.[13] 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

- Eventual consistency (BASE: basically available)
- CAP theorem
- Concurrency control
- Java Transaction API
- Open Systems Interconnection
- Transactional NTFS
- Two-phase commit protocol
- CRUD

References

1. Haerder, T.; Reuter, A. (1983). "Principles of transaction-oriented database recovery". *ACM Computing Surveys*. **15** (4): 287. doi:10.1145/289.291 (https://doi.org/10.1145%2F289.291). S2CID 207235758 (https://api.semanticscholar.org/CorpusID:207235758).
2. Gray, Jim (September 1981). "The Transaction Concept: Virtues and Limitations" (http://research.microsoft.com/~gray/papers/theTransactionConcept.pdf) (PDF). *Proceedings of the 7th International Conference on Very Large Databases*. Cupertino, California: Tandem Computers. pp. 144–154. Retrieved March 27, 2015.
3. Gray, Jim; Reuter, Andreas (1993). *Distributed Transaction Processing: Concepts and Techniques* (https://archive.org/details/transactionproce0000gray). Morgan Kaufmann. ISBN 1-55860-190-2.
4. "ACID vs BASE Databases - Difference Between Databases - AWS" (https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/). *Amazon Web Services, Inc*. Retrieved 2025-03-24.
5. "ACID vs BASE Databases - Difference Between Databases - AWS" (https://aws.amazon.com/compare/the-difference-between-acid-and-base-database/). *Amazon Web Services, Inc*. Retrieved 2025-03-24.