ACID represents four important elements in the database that ensure that the transaction works as we expect. They are atomicity, consistency, isolation, and durability.

Basically, almost every mainstream database software will make their transaction meet these four requirements. Although the way of compliance may be slightly different, the general target they want to achieve is the same.

When we are facing the so-called distributed system design problem, in the database layer, ACID is a very important basic concept. Without familiarity with these basic concepts, it is difficult to design a reliable database system that can handle large traffic (high concurrency problem).

First, please imagine a classic case: Our system initiates a transaction that transfers money from bank account `A` to bank account `B`. Later, we will use this case as an example to help you understand the concept of ACID.

Atomicity: The Smallest Unit

If we want to use transaction, then we should regard a group of queries as the smallest execution unit in SQL. That is to say, even if there are multiple statements in a transaction, we have to treat them as one. The reason is if a bug is triggered when we are half way through a group of queries for updating anything, the whole batch of updates should be discarded..

Through the design of atomicity, the database can ensure that SQL statements are completely bound together and will not be split.

Otherwise, it would be terrible if a crash event caused a large transaction to fail halfway through execution, and the execution result of SQL was cut into two parts, where the first half was executed, but the second half wasn't.

Example

Take the case of this chapter as an example when we want to perform a bank transaction between accounts `A` and `B`. Without atomicity, it is possible that after the database finishes the deduction of account `A`, it crashes without handling the money transfer to `B`. In this case, it is a disaster because `A` has been debited, but `B` will not receive the money.

That's because, without atomicity implemented by the database, the execution results of the withdrawal and deposit statements are split and not treated as a complete set.

Consistency: Always Make Data Consistent

In the database, we will set many diverse constraints. In the case of many transaction operation successes and failures, a database with the ability to remain consistent can ensure that these constraints are satisfied.

Therefore, consistency focuses on a key point, keep the data in the correct appearance. We learned how to use the `ROLLBACK` keyword in the previous chapter to ensure the correctness of the data and return it to the correct state (before the transaction started). Consistency helps us to ensure that data remains in the correct state.

Example

In the case of this chapter, with consistency, we can use the `ROLLBACK` keyword to go back to the original if there is any crash event that happens during the execution.

Isolation: Transactions Do Not Affect Each Other

Remember we mentioned at the beginning that the concept of ACID is a must-have basic knowledge to undertake systems with a large amount of traffic? Isolation is used to solve data problems when traffic increases; it ensures that transactions do not affect each other.

When many transactions occur, some transactions that are not exempted may need to modify the same record, which will make the problem become very complicated, and it will be a challenge to determine which version is correct.

Isolation requires the database to ensure that transactions will not affect each other to a certain extent. To understand to what extent, you can refer to Isolation Levels. The implementation is similar across different databases.

Example

Consider a public account `A` where different people can withdraw money from `A` at the same time. Isolation can ensure that the data columns that are required for both are locked by one first, and the other transaction is queued until the record has been unlocked.

> Locked means the database prohibits any other transaction from modifying the locked record.

But be careful, an inappropriate lock mechanism may cause a deadlock error and cause the database to crash, so still pay attention to the possibility of related problems.

Durability: Always Exist

In addition to software crashes, there may also be hardware-level failures such as power outages in the real world. For such events, a database with durability capability can ensure that as long as a result has been committed, it must exist in the hard disk data permanently.

This is a relatively simple concept, but it is vital that we ensure the data will not be lost.

Example

A temporary power outage in the data center where our transaction database is stored resulted in a few minutes of application downtime. After recovery, we find many data that should have executed "commit" are lost. This is an example of the problem of no `durability`.