WIKIPEDIA
The Free Encyclopedia

# Database transaction

A **database transaction** symbolizes a unit of work, performed within a database management system (or similar system) against a database, that is treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database. Transactions in a database environment have two main purposes:

1. To provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure. For example: when execution prematurely and unexpectedly stops (completely or partially) in which case many operations upon a database remain uncompleted, with unclear status.
2. To provide isolation between programs accessing a database concurrently. If this isolation is not provided, the programs' outcomes are possibly erroneous.

In a database management system, a transaction is a single unit of logic or work, sometimes made up of multiple operations. Any logical calculation done in a consistent mode in a database is known as a transaction. One example is a transfer from one bank account to another: the complete transaction requires subtracting the amount to be transferred from one account and adding that same amount to the other.

A database transaction, by definition, must be atomic (it must either be complete in its entirety or have no effect whatsoever), consistent (it must conform to existing constraints in the database), isolated (it must not affect other transactions) and durable (it must get written to persistent storage).[1] Database practitioners often refer to these properties of database transactions using the acronym ACID.

## Purpose

Databases and other data stores which treat the integrity of data as paramount often include the ability to handle transactions to maintain the integrity of data. A single transaction consists of one or more independent units of work, each reading and/or writing information to a database or other data store. When this happens it is often important to ensure that all such processing leaves the database or data store in a consistent state.

Examples from double-entry accounting systems often illustrate the concept of transactions. In double-entry accounting every debit requires the recording of an associated credit. If one writes a check for $100 to buy groceries, a transactional double-entry accounting system must record the following two entries to cover the single transaction:

1. Debit $100 to Groceries Expense Account
2. Credit $100 to Checking Account

A transactional system would make both entries pass or both entries would fail. By treating the recording of multiple entries as an atomic transactional unit of work the system maintains the integrity of the data recorded. In other words, nobody ends up with a situation in which a debit is recorded but no associated credit is recorded, or vice versa.

# Transactional databases

A **transactional database** is a DBMS that provides the ACID properties for a bracketed set of database operations (begin-commit). Transactions ensure that the database is always in a consistent state, even in the event of concurrent updates and failures.[2] All the write operations within a transaction have an all-or-nothing effect, that is, either the transaction succeeds and all writes take effect, or otherwise, the database is brought to a state that does not include any of the writes of the transaction. Transactions also ensure that the effect of concurrent transactions satisfies certain guarantees, known as isolation level. The highest isolation level is serializability, which guarantees that the effect of concurrent transactions is equivalent to their serial (i.e. sequential) execution.

Most modern relational database management systems support transactions. NoSQL databases prioritize scalability along with supporting transactions in order to guarantee data consistency in the event of concurrent updates and accesses.

In a database system, a transaction might consist of one or more data-manipulation statements and queries, each reading and/or writing information in the database. Users of database systems consider consistency and integrity of data as highly important. A simple transaction is usually issued to the database system in a language like SQL wrapped in a transaction, using a pattern similar to the following:

1. Begin the transaction.
2. Execute a set of data manipulations and/or queries.
3. If no error occurs, then commit the transaction.
4. If an error occurs, then roll back the transaction.

A transaction commit operation persists all the results of data manipulations within the scope of the transaction to the database. A transaction rollback operation does not persist the partial results of data manipulations within the scope of the transaction to the database. In no case can a partial transaction be committed to the database since that would leave the database in an inconsistent state.

Internally, multi-user databases store and process transactions, often by using a transaction ID or XID.

There are multiple varying ways for transactions to be implemented other than the simple way documented above. Nested transactions, for example, are transactions which contain statements within them that start new transactions (i.e. sub-transactions). *Multi-level transactions* are a variant of nested transactions where the sub-transactions take place at different levels of a layered system architecture (e.g., with one operation at the database-engine level, one operation at the operating-system level).[3] Another type of transaction is the compensating transaction.

### In SQL

Transactions are available in most SQL database implementations, though with varying levels of robustness. For example, MySQL began supporting transactions from early version 3.23, but the InnoDB storage engine was not default before version 5.5. The earlier available storage engine, MyISAM does not support transactions.

A transaction is typically started using the command `BEGIN` (although the SQL standard specifies `START TRANSACTION`). When the system processes a `COMMIT` statement, the transaction ends with successful completion. A `ROLLBACK` statement can also end the transaction, undoing any work performed since `BEGIN`. If autocommit was disabled with the start of a transaction, autocommit will also be re-enabled with the end of the transaction.

One can set the isolation level for individual transactional operations as well as globally. At the highest level (`READ COMMITTED`), the result of any operation performed after a transaction has started will remain invisible to other database users until the transaction has ended. At the lowest level (`READ UNCOMMITTED`), which may occasionally be used to ensure high concurrency, such changes will be immediately visible.

## Object databases

Relational databases are traditionally composed of tables with fixed-size fields and records. Object databases comprise variable-sized blobs, possibly serializable or incorporating a mime-type. The fundamental similarities between Relational and Object databases are the start and the commit or rollback.

After starting a transaction, database records or objects are locked, either read-only or read-write. Reads and writes can then occur. Once the transaction is fully defined, changes are committed or rolled back atomically, such that at the end of the transaction there is no inconsistency.

## Distributed transactions

Database systems implement distributed transactions[4] as transactions accessing data over multiple nodes. A distributed transaction enforces the ACID properties over multiple nodes, and might include systems such as databases, storage managers, file systems, messaging systems, and other data managers. In a distributed transaction there is typically an entity coordinating all the process to ensure that all parts of the transaction are applied to all relevant systems. Moreover, the integration of Storage as a Service (StaaS) within these environments is crucial, as it offers a virtually infinite pool of storage resources, accommodating a range of cloud-based data store classes with varying availability, scalability, and ACID properties. This integration is essential for achieving higher availability, lower response time, and cost efficiency in data-intensive applications deployed across cloud-based data stores.[5]