Engineer

Transaction

a group of database changes that succeed or fail as one

What it is

A transaction is a group of changes that a database applies as one unit: either all of them take effect, or none of them does. People write database transaction when they need to keep it apart from a payment, which shares the word. The PostgreSQL tutorial describes a transaction as something that "bundles multiple steps into a single, all-or-nothing operation", and nothing in between is visible to anyone else.

In PostgreSQL a transaction opens with `BEGIN` and ends with `COMMIT`, which makes its changes permanent and visible to other sessions, or with `ROLLBACK`, which throws them away. A statement sent on its own still runs inside one: PostgreSQL wraps it in an implicit `BEGIN` and, if it succeeds, an implicit `COMMIT`. A long transaction can set a `SAVEPOINT` and later return to it with `ROLLBACK TO`, discarding the work done since while keeping everything before it.

The guarantees a transaction gives are summed up as ACID. Atomicity means a transaction completes as one unit or not at all, and no half-finished result survives a crash. Consistency means the constraints declared on the data hold once it commits, and a transaction that would leave one broken is rolled back instead. Isolation means its changes stay invisible to concurrent transactions until it commits. Durability means a committed change survives a crash of the server.

Isolation comes in levels, and the SQL standard defines four: Read Uncommitted, Read Committed, Repeatable Read and Serializable. PostgreSQL starts every transaction at Read Committed unless told otherwise, and its Read Uncommitted behaves exactly like Read Committed. At Read Committed each statement sees the data committed before that statement began, so two queries in the same transaction may see different numbers. At Repeatable Read the whole transaction works from one snapshot, taken when its first statement other

than a transaction-control statement runs. Serializable goes further and makes concurrent transactions produce the same result as if they had run one after another.

Use the concept in a decision

Put “What is a transaction?” to work in a real decision. We can help relate the concept to the system you are building or buying. A structure that matches how the business actually works, so the questions you need answered later are still possible to ask.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

Almost every action worth recording touches more than one row. An order is a row in one table, a decrement in another and an entry in a ledger. Without a transaction, a crash, a timeout or a failed constraint between the second and the third statement leaves an order whose stock was never taken, or stock taken for an order that does not exist. Someone then finds it in a report and repairs it by hand. With a transaction that half state never exists for anyone to see, and the database, not the application, is what guarantees it.

The second reason is concurrency. Two requests that arrive together run side by side, and the isolation level decides what each one sees while locks decide which one waits. This is where teams are most often surprised: wrapping code in `BEGIN` and `COMMIT` does not on its own stop two requests from reading the same balance and both spending it. At Read Committed each statement reads what was committed when it started, so a check made in one statement and a write made in the next can be separated by another session’s commit. The check has to lock the row first or become part of the write.

Transactions also have a cost, and it is paid in time. Row locks taken by `UPDATE`, `DELETE` and `SELECT ... FOR UPDATE` normally last until the transaction ends; rolling back to a savepoint releases locks acquired after that savepoint. A session that wants a conflicting lock must wait unless a timeout, cancellation or deadlock resolution stops it. An open transaction also stops `VACUUM` from removing old row versions it might still need to see, so a transaction left open for hours lets tables bloat. The design question is never only whether to use a transaction. It is where its edges go and how little happens between them.

Finally, an ordinary local database transaction does not by itself include external effects. A call to a payment provider, an email or a message published to a separate queue is not undone by rolling back the database. Coordinating other systems needs an additional design; a surrounding `BEGIN` is not enough. Deciding what sits inside the boundary, and how the outside world hears about a commit, is most of the work of designing one.

What it looks like in practice

A member of a loyalty scheme spends 250 points on a cinema voucher. Three things have to happen together: the wallet loses the points, the ledger records why, and the redemption is created. Written as one transaction it looks like this:

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- no row back: not enough points, so ROLLBACK and stop here

INSERT INTO points_entry (member_id, delta, reason)
VALUES (42, -250, 'redeemed: cinema voucher');

INSERT INTO redemption (member_id, reward_id)
VALUES (42, 7);

COMMIT;
```

The `UPDATE` takes a row lock on member 42's wallet. Until `COMMIT`, any other session that reads the wallet still sees the old balance, because row locks do not block readers and uncommitted changes are invisible. A second redemption for the same member, sent from another tab a moment later, reaches its own `UPDATE` and waits. When the first transaction commits, PostgreSQL at Read Committed evaluates the second one's `WHERE` clause again against the row as it now stands. If only 100 points are left, the condition fails, no row comes back, and the second request rolls back without having written anything. If instead the last `INSERT` fails, say because reward 7 has been withdrawn and a foreign key refuses it, the client rolls back and the points are exactly where they were.

This is the shape of the loyalty wallet with leagues and badges. Every change to a wallet there takes a row lock on it before reading it, so a second request waits for the first to finish instead of racing it, and because the wallet lives in the same database as everything else, a balance and what it paid for are committed together or not at all. The page is open about the price: row locking is the slower answer, and members wait their turn rather than spend the same points twice.

The note on selling the last ticket exactly once puts the same tool under a flash sale. Its fix makes the stock check and the decrement one conditional `UPDATE`, because a lock held while the application thinks between a `SELECT` and an `UPDATE` makes the next buyer wait for every millisecond of it. First on its list for the rest of the work is keeping everything external out of the transaction: a payment call, an HTTP request or a published message belongs before it or after it.

Where it goes wrong

A session sits idle in a transaction and everything behind it waits. An application opens a transaction, updates a row, and then waits for a slow API, for a user's next click, or for a worker that crashed without ever sending `COMMIT`. Its locks stay, and with timeouts disabled PostgreSQL makes a session that wants a conflicting lock wait indefinitely unless it detects a deadlock or the wait is cancelled. The documentation calls holding a transaction open while waiting for user input a bad idea. `pg_stat_activity` shows these sessions with the state `idle in transaction`, and `xact_start` says when the transaction began:

```
SELECT pid, now() - xact_start AS open_for, state, query
FROM pg_stat_activity
WHERE state LIKE 'idle in transaction%'
ORDER BY xact_start;
```

Move the slow work out of the transaction, and give the server a limit so a forgotten one cannot hold locks for hours. `idle_in_transaction_session_timeout` ends any session left idle inside an open transaction for longer than its value. Its default is 0, which disables it, so it has to be set, for example for the application's role with `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';`.

Two requests both spend the same balance. The code reads a value in one statement, decides in the application, and writes the result in another. Two sessions read the same number, both pass the check, and both write, so one change silently overwrites the other. A surrounding `BEGIN` does not prevent it at Read Committed. Make the check part of the write with a condition in the `UPDATE`, as in the example above, or lock the row before reading it with `SELECT ... FOR UPDATE`, which makes the second session wait for the first.

Errors appear under load at Repeatable Read or Serializable. At these levels PostgreSQL refuses a transaction that would break the level's promise and reports a serialization failure, SQLSTATE `40001`, with messages such as `could not serialize access due to concurrent update`. This is the design working, not a bug, and the documentation states that applications using these levels must be prepared to retry. Catch `40001` and run the whole transaction again from `BEGIN`, including its reads, since the decision it made was based on data that has since changed. Retrying only the statement that failed repeats the mistake.

Two transactions deadlock. One locks order 1 and then order 2, another locks order 2 and then order 1, and each waits for the other. PostgreSQL detects the cycle and aborts one of the transactions so the other can finish, and which one it picks is hard to predict. The documented defence is to have every code path acquire locks in the same order, for example by updating rows in ascending order of their primary key, and to retry a transaction that was aborted for a deadlock.

An email goes out for an order that was rolled back. The application sends a confirmation or publishes an event from inside the transaction, and then a later statement fails. The rollback removes the order, but nothing can unsend the email. Record the message in an outbox table in the same transaction as the change it describes, and let a separate process send it once the commit has happened. The message then exists exactly when the order does.

Further reading

- Transactions — the PostgreSQL tutorial chapter: all-or-nothing behaviour, implicit transactions and savepoints, shown on a bank transfer.
- Transaction Isolation — the four standard levels, what each one sees in PostgreSQL, and the serialization failures that call for a retry.
- Explicit Locking — row and table locks, how long they are held, and how deadlocks are detected and avoided.

- Client Connection Defaults — `default_transaction_isolation`, `idle_in_transaction_session_timeout`, `lock_timeout` and the other limits on a session.
- Routine Vacuuming — why old row versions stay while any transaction might still see them, and what long open transactions do to cleanup.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a transaction?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/transaction/