Engineer

Connection pool

a set of open database connections kept ready for reuse

What it is

A connection pool is a set of open database connections that an application keeps and lends out again, instead of opening a new connection for every piece of work and closing it afterwards. Code checks a connection out of the pool, runs its statements on it and hands it back, and the next caller that needs one gets the same connection, already open. SQLAlchemy's documentation describes pooling as "a standard technique used to maintain long running connections in memory for efficient reuse". The same page gives the pool a second job: it manages how many connections the application may use at the same time, so a burst of traffic waits for a free connection instead of opening more.

A pool can live in two places. Inside the application it is part of the database library: every SQLAlchemy `Engine` comes with one, and Django can keep one in each process for PostgreSQL. Outside the application it is a separate program that sits between the clients and the database server. PgBouncer is such a pooler for PostgreSQL: applications connect to it as if it were the database, and it lends them server connections from pools of its own. The two can be combined, and Django's database reference has a section on running behind a transaction pooler such as PgBouncer.

Why a connection is worth keeping shows on the server side. PostgreSQL gives each client a process of its own: a supervisor process called the postmaster listens for incoming connections, starts a new server process for every connection request, and each client is connected to exactly one of those processes. Opening a connection therefore means starting a process on the database server, and a pool pays that price once and then reuses the result.

Use the concept in a decision

Put “What is a connection pool?” to work in a real decision. We can help relate the concept to the system you are building or buying. Finding the queries and the missing indexes behind a system that has got heavier every year, and fixing the ones that matter.

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. 30 days of electricity data in 180 ms — 180 ms to aggregate a month of meter readings

30 days of electricity data in 180 ms Get in touch →

Why it matters

The first reason is the cost of connecting. Django’s documentation makes the case for keeping connections in plain terms: a persistent connection avoids the overhead of establishing a new one in each HTTP request. Its default does the opposite, though. `CONN_MAX_AGE` is 0, which closes the connection at the end of every request; a positive number of seconds keeps it open for reuse, and `None` keeps it open without a limit. That is the simplest form of reuse, one connection per thread that outlives the request, and a pool is the general form of it.

The second reason is the limit on the server. PostgreSQL’s `max_connections` is typically 100 by default, may be lower if the kernel settings will not support it, and can only be changed at server start. Three of those slots are reserved for superusers by default. Raising the number is not free either: PostgreSQL sizes certain resources, shared memory among them, directly from `max_connections`. Without a pool, every worker holds a connection of its own, and Django’s documentation spells out what follows: because each thread keeps its own connection, the database has to accept at least as many simultaneous connections as there are worker threads. A pool turns that into a number the team chooses and can add up.

The third reason is hygiene. A pool is the one place every connection passes through between two borrowers, so it is where a connection gets cleaned and tested. When a connection comes back, SQLAlchemy’s pool calls `rollback()` on it, which releases any transactional state, locks included, so the next borrower does not inherit a half-finished transaction. In PgBouncer’s default session-pooling mode, its `server_reset_query`, `DISCARD ALL` by default, runs when a server connection is released and before another client receives it. And a pool can test a connection before lending it out, which matters for connections that have sat idle for a while.

What it looks like in practice

In SQLAlchemy 2.1 every `Engine` gets a `QueuePool` by default, with SQLite's in-memory database as the exception. Its `pool_size` is 5 and its `max_overflow` is 10, so one engine can hold up to 15 connections at once, and a caller that finds them all in use waits `pool_timeout`, 30 seconds by default, before giving up. The settings are keyword arguments to `create_engine()`:

```
from sqlalchemy import create_engine

# At most ten connections for this engine: ten kept open, none beyond.
engine = create_engine(
    "postgresql+psycopg2://me@localhost/mydb",
    pool_size=10,
    max_overflow=0,
    pool_timeout=10,  # wait ten seconds for a free connection, then give up
    pool_pre_ping=True,  # test each connection as it is checked out
    pool_recycle=3600,  # replace connections that are over an hour old
)
```

With `pool_pre_ping`, the pool checks at checkout that the connection is still usable, through the driver's own ping or a statement equivalent to `SELECT 1`. `pool_recycle` is meant for servers that close connections left unused for a certain time: with 3600, any connection open for more than an hour is replaced at its next checkout.

Django 6.1 has a pool of its own for PostgreSQL. It works only with the psycopg driver and needs the `psycopg[pool]` package, and it is switched on with `"OPTIONS": {"pool": True}` in the database settings, or with a dictionary of options for psycopg's `ConnectionPool`. The pool reuses connections across threads, and each process keeps a separate pool for each database alias. `CONN_MAX_AGE` must stay at 0 when pooling is on; the lifetime of a pooled connection is set with the pool's `max_lifetime` option instead. Under ASGI, Django's documentation advises turning persistent connections off and using this built-in pool instead.

PgBouncer's `pool_mode` decides when a server connection can go to another client. In session mode, the default, the client keeps it until it disconnects. In transaction mode, a client has a server connection only while a transaction runs, so a few dozen server connections can serve many more clients. Statement mode releases the connection after every query and does not allow transactions that

span several statements. Each pool holds up to `default_pool_size` server connections per user and database pair, 20 by default, and PgBouncer accepts up to `max_client_conn` client connections, 100 by default.

The company's own notes put the pool early in the order of work. Optimise the database before you shard it has the reader count the active sessions in `pg_stat_activity` against the CPU cores: when many times more are active than there are cores, the server spends its time switching between them, and a connection pool fixes that before any hardware does. Selling the last ticket exactly once applies the same idea to a flash sale: a hundred thousand buyers should become a few dozen connections through PgBouncer, not thousands of server processes waiting for one lock.

Where it goes wrong

The pools add up to more than the server allows. Each pool counts only its own connections. At SQLAlchemy's defaults one engine can reach 15, so twelve processes can ask for 180 from a server whose default `max_connections` is typically 100, three of them reserved for superusers, and Django keeps a separate pool for every database alias in every process. Add up the maximum of every pool that points at the database, across every process of every application and every scheduled job, and keep the sum below `max_connections` minus the reserved slots. When it cannot fit, put PgBouncer in front and let it hold the number of server connections.

A connection dies while it waits in the pool. A connection that sat idle can be closed on the other side without the pool knowing, and the next borrower then fails on its first statement. Turn on `pool_pre_ping` in SQLAlchemy or `CONN_HEALTH_CHECKS` in Django, which tests a reused connection once per request, and set `pool_recycle` or psycopg's `max_lifetime` below the time after which the server closes idle connections. Neither covers a connection dropped in the middle of a transaction: SQLAlchemy's documentation says plainly that pre-ping does not handle that case, so the application has to retry the work itself.

A forked worker inherits the parent's connections. A web server or a job runner that creates the engine and then forks its workers leaves every child holding the same pooled connections as the parent. The SQLAlchemy documentation calls it critical that pooled connections are not shared with a forked process, because both processes would then talk over the same network connection. Follow its recommended approach and call

`engine.dispose(close=False)` in each child's initializer, so the child starts with an empty pool and leaves the parent's connections alone; the alternatives are `NullPool`, which opens and closes a connection for every use, or calling `engine.dispose()` before forking.

Transaction pooling breaks session state. In PgBouncer's transaction mode, two transactions from the same client can run on different server connections, so anything a client sets up for its session is not guaranteed to be there for its next transaction. PgBouncer's feature list marks `SET`, `LISTEN`, cursors declared `WITH HOLD`, SQL-level `PREPARE` and session-level advisory locks as never supported in that mode, and is open that the mode, by its nature, breaks some of the assumptions clients make about the server. Keep session state out of code that goes through a transaction pooler, and send the clients that need it to a pool in session mode. In Django, set `DISABLE_SERVER_SIDE_CURSORS` to `True` for a database behind a transaction pooler, or run those queries inside `atomic()`.

Further reading

- Connection Pooling — `QueuePool` and its defaults, disconnect handling, reset on return and pools under fork in SQLAlchemy 2.1.
- PgBouncer features — the session, transaction and statement modes, and which PostgreSQL features each one supports.
- PgBouncer configuration — `pool_mode`, `default_pool_size`, `max_client_conn` and the timeouts, with their defaults.
- How Connections Are Established — PostgreSQL's process per user model and the postmaster.
- Connections and Authentication — `max_connections` and the connection slots reserved for superusers and other roles.
- Databases — persistent connections, the PostgreSQL connection pool and transaction pooling, in Django's reference.

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 connection pool?” 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/connection-pool/