Engineer

Пул з'єднань

набір відкритих з'єднань, готових до повторного використання

Що це таке

Пул з’єднань — це набір відкритих з’єднань із базою даних, які застосунок тримає й позичає знову, замість того щоб відкривати нове з’єднання для кожної дії та закривати його після неї. Його ще називають пулом підключень. Код бере з’єднання з пулу, виконує на ньому свої інструкції та повертає його, і наступний, кому воно потрібне, отримує те саме з’єднання, уже відкрите. Документація SQLAlchemy описує пулінг як стандартний прийом, що тримає довгоживучі з’єднання в пам’яті для ефективного повторного використання. Та сама сторінка дає пулу ще одну роль: він керує тим, скільки з’єднань застосунок може використовувати одночасно, тож раптовий сплеск трафіку чекає на вільне з’єднання, а не відкриває нові.

Пул може жити у двох місцях. Усередині застосунку він є частиною бібліотеки для роботи з базою: кожен `Engine` у SQLAlchemy має власний пул, а Django може тримати по одному в кожному процесі для PostgreSQL. Поза застосунком це окрема програма, що стоїть між клієнтами й сервером бази даних. PgBouncer — саме такий пулер для PostgreSQL: застосунки під’єднуються до нього так, ніби це база даних, а він позичає їм серверні з’єднання з власних пулів. Ці два варіанти можна поєднувати, і в довіднику Django про бази даних є розділ про роботу за транзакційним пулером на кшталт PgBouncer.

Чому з’єднання варто зберігати, видно на боці сервера. PostgreSQL дає кожному клієнтові окремий процес: головний процес, postmaster, слухає вхідні з’єднання й запускає новий серверний процес на кожен запит на з’єднання, і кожен клієнт під’єднаний рівно до одного з цих процесів. Отже, відкрити з’єднання означає запустити процес на сервері бази даних, а пул платить цю ціну один раз і далі використовує результат повторно.

Застосуйте поняття до рішення

Застосуйте «Що таке пул з'єднань?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Пошук запитів і відсутніх індексів у системі, що рік за роком важчає, та виправлення тих, що справді мають значення.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. 30 днів даних про електроенергію за 180 мс — 180 ms на агрегацію місяця показань лічильників

30 днів даних про електроенергію за 180 мс Зв'язатися з нами →

Чому це важливо

Перша причина — ціна під’єднання. Документація Django каже про це прямо: постійне з’єднання позбавляє від витрат на встановлення нового в кожному HTTP-запиті. Проте типове налаштування робить навпаки. `CONN_MAX_AGE` дорівнює 0, і з’єднання закривається наприкінці кожного запиту; додатне число секунд тримає його відкритим для повторного використання, а `None` — без обмеження. Це найпростіша форма повторного використання, одне з’єднання на потік, яке живе довше за запит, а пул — її загальна форма.

Друга причина — межа на сервері. Типове значення `max_connections` у PostgreSQL зазвичай становить 100, може бути меншим, якщо налаштування ядра цього не витримують, і змінюється лише під час запуску сервера. Три місця з них типово зарезервовано для суперкористувачів. Підняти це число теж не безкоштовно: PostgreSQL визначає розмір деяких ресурсів, зокрема спільної пам’яті, безпосередньо з `max_connections`. Без пулу кожен робочий процес тримає власне з’єднання, і документація Django пояснює наслідок: оскільки кожен потік має своє з’єднання, база даних мусить приймати щонайменше стільки одночасних з’єднань, скільки є робочих потоків. Пул дає команді змогу самій визначати кількість з’єднань і підсумовувати її для всіх пулів.

Третя причина — гігієна. Пул — це єдине місце, через яке кожне з’єднання проходить між двома позичальниками, тож саме тут його прибирають і перевіряють. Коли з’єднання повертається, пул SQLAlchemy викликає на ньому `rollback()`, і це звільняє весь транзакційний стан разом із блокуваннями, тож наступний позичальник не успадкує недороблену транзакцію. У типовому режимі PgBouncer із пулом сеансів його `server_reset_query`, типово `DISCARD ALL`, виконується після звільнення серверного з’єднання й перед тим, як його отримає інший клієнт. А ще пул може перевірити з’єднання, перш ніж його позичити, і це важливо для з’єднань, які довго простоювали.

Як це виглядає на практиці

У SQLAlchemy 2.1 кожен `Engine` типово отримує `QueuePool`, окрім бази даних SQLite у пам'яті. Його `pool_size` дорівнює 5, а `max_overflow` — 10, тож один engine може мати до 15 з'єднань одночасно, а виклик, який застає їх усі зайнятими, чекає `pool_timeout`, типово 30 секунд, і лише тоді здається. Ці параметри передають як іменовані аргументи до `create_engine()`:

```
from sqlalchemy import create_engine

# Щонайбільше десять з'єднань на цей engine: усі відкриті, понад них жодного.
engine = create_engine(
    "postgresql+psycopg2://me@localhost/mydb",
    pool_size=10,
    max_overflow=0,
    pool_timeout=10,  # чекати на вільне з'єднання десять секунд, далі здатися
    pool_pre_ping=True,  # перевіряти кожне з'єднання під час видачі
    pool_recycle=3600,  # замінювати з'єднання, старші за годину
)
```

Із `pool_pre_ping` пул під час видачі перевіряє, що з'єднання ще придатне, через власний ping драйвера або інструкцію, рівнозначну `SELECT 1`. `pool_recycle` призначений для серверів, які закривають з'єднання, що певний час не використовувалися: зі значенням 3600 кожне з'єднання, відкрите понад годину, замінюється під час наступної видачі.

Django 6.1 має власний пул для PostgreSQL. Він працює лише з драйвером psycopg, потребує пакета `psycopg[pool]`, а вмикають його через `"OPTIONS": {"pool": True}` у налаштуваннях бази даних або словником параметрів для `ConnectionPool` із psycopg. Пул повторно використовує з'єднання між потоками, і кожен процес має окремий пул для кожного псевдоніма бази даних. `CONN_MAX_AGE` має лишатися 0, коли пул увімкнено; час життя з'єднання в пулі натомість задає параметр пулу `max_lifetime`. Для ASGI документація Django радить вимкнути постійні з'єднання й користуватися цим вбудованим пулом.

Параметр PgBouncer `pool_mode` визначає, коли серверне з'єднання може перейти до іншого клієнта. У режимі `session`, типовому, клієнт тримає з'єднання, доки не від'єднається. У режимі `transaction` клієнт має серверне з'єднання лише поки триває транзакція, тож кілька десятків серверних з'єднань можуть обслуговувати значно більше клієнтів. Режим `statement`

звільняє з'єднання після кожного запиту й не дозволяє транзакцій із кількох інструкцій. Кожен пул вміщує до `default_pool_size` серверних з'єднань на пару користувача й бази даних, типово 20, а PgBouncer приймає до `max_client_conn` клієнтських з'єднань, типово 100.

Власні нотатки компанії ставлять пул на ранній крок. Оптимізуйте базу, перш ніж її шардувати пропонує порахувати активні сесії в `pg_stat_activity` і порівняти з кількістю ядер CPU: коли активних у багато разів більше, ніж ядер, сервер витрачає час на перемикання між ними, і пул з'єднань виправить це раніше, ніж будь-яке залізо. Продати останній квиток рівно один раз застосовує ту саму думку до флеш-розпродажу: сто тисяч покупців мають перетворитися на кілька десятків з'єднань через PgBouncer, а не на тисячі серверних процесів, що чекають на одне блокування.

Де виникають проблеми

Пули разом просять більше з'єднань, ніж дозволяє сервер. Кожен пул рахує лише власні з'єднання. За типових значень SQLAlchemy один engine може дійти до 15, тож дванадцять процесів можуть попросити 180 у сервера, де `max_connections` зазвичай типово дорівнює 100, а три з них зарезервовано для суперкористувачів; до того ж Django має окремий пул для кожного псевдоніма бази даних у кожному процесі. Складіть максимум кожного пулу, що дивиться на цю базу даних, по всіх процесах усіх застосунків і всіх запланованих завдань, і тримайте суму нижче за `max_connections` мінус зарезервовані місця. Якщо не вміщається, поставте попереду PgBouncer і дайте йому тримати кількість серверних з'єднань.

З'єднання вмирає, поки чекає в пулі. З'єднання, яке простоювало, можуть закрити з іншого боку так, що пул про це не знає, і наступний позичальник тоді падає на першій же інструкції. Увімкніть `pool_pre_ping` у SQLAlchemy або `CONN_HEALTH_CHECKS` у Django, яке перевіряє повторно використане з'єднання раз на запит, і задайте `pool_recycle` або `max_lifetime` у psycopg меншим за час, після якого сервер закриває невикористовувані з'єднання. Жоден із цих засобів не рятує з'єднання, що обривається посеред транзакції: документація SQLAlchemy прямо каже, що pre-ping цього випадку не обробляє, тож застосунок мусить сам повторити роботу.

Робочий процес, створений через fork, успадковує з'єднання батьківського. Вебсервер або обробник завдань, що створює свій engine, а потім запускає робочі процеси через fork, залишає кожному дочірньому

процесу ті самі з'єднання з пулу, що й у батьківського. Документація SQLAlchemy називає критично важливим не ділити з'єднання з пулу з процесом, створеним через fork, бо обидва процеси тоді говоритимуть через одне й те саме мережеве з'єднання. Дотримуйтеся рекомендованого підходу й викликайте `engine.dispose(close=False)` в ініціалізаторі кожного дочірнього процесу, щоб він починав із порожнього пулу й не чіпав з'єднань батьківського. Альтернативи — `NullPool`, який відкриває й закриває з'єднання для кожного використання, або виклик `engine.dispose()` перед fork.

Транзакційний пулінг ламає стан сесії. У режимі `transaction` у PgBouncer дві транзакції одного клієнта можуть виконуватися на різних серверних з'єднаннях, тож усе, що клієнт налаштовує для своєї сесії, не обов'язково буде на місці в його наступній транзакції. Перелік можливостей PgBouncer позначає `SET`, `LISTEN`, курсори, оголошені з `WITH HOLD`, `PREPARE` на рівні SQL і рекомендаційні блокування на рівні сесії як ніколи не підтримувані в цьому режимі й відверто визнає, що режим за своєю природою порушує деякі припущення клієнтів щодо сервера. Тримайте стан сесії поза кодом, що йде через транзакційний пулер, а клієнтів, яким він потрібен, спрямовуйте до пулу в режимі `session`. У Django для бази даних за транзакційним пулером встановіть `DISABLE_SERVER_SIDE_CURSORS` у `True` або виконуйте ці запити всередині `atomic()`.

Що почитати далі

- Connection Pooling — `QueuePool` і його типові значення, обробка розірваних з'єднань, скидання під час повернення та пули після fork у SQLAlchemy 2.1.
- PgBouncer features — режими session, transaction і statement та можливості PostgreSQL, які підтримує кожен із них.
- PgBouncer configuration — `pool_mode`, `default_pool_size`, `max_client_conn` і тайм-аути з їхніми типовими значеннями.
- How Connections Are Established — модель PostgreSQL з окремим процесом на кожного користувача і процес postmaster.
- Connections and Authentication — `max_connections` і місця для з'єднань, зарезервовані для суперкористувачів та інших ролей.
- Databases — постійні з'єднання, пул для PostgreSQL і транзакційний пулінг у довіднику Django.

Словник

Обирайте за фактичними умовами

Надайте репрезентативний набір даних, запитання до нього та поточне вузьке місце. Узгодимо вимірювання якості, пропускної здатності та відновлення.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке пул з'єднань?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/connection-pool/