Engineer

Tilkoblingspool

et sett åpne databasetilkoblinger klare til gjenbruk

Hva det er

En tilkoblingspool er et sett åpne databasetilkoblinger som en applikasjon holder på og låner ut igjen, i stedet for å åpne en ny tilkobling for hver arbeidsoppgave og lukke den etterpå. Koden låner en tilkobling fra poolen, kjører setningene sine på den og leverer den tilbake, og den neste som trenger en, får den samme tilkoblingen, allerede åpen. SQLAlchemys dokumentasjon beskriver pooling som en standardteknikk for å holde langlivede tilkoblinger i minnet slik at de kan gjenbrukes effektivt. Samme side gir poolen en oppgave til: den styrer hvor mange tilkoblinger applikasjonen kan bruke samtidig, slik at en plutselig trafikktopp venter på en ledig tilkobling i stedet for å åpne flere.

En pool kan ligge to steder. Inne i applikasjonen er den en del av databasebiblioteket: hver `Engine` i SQLAlchemy har en, og Django kan holde en i hver prosess for PostgreSQL. Utenfor applikasjonen er den et eget program som står mellom klientene og databaseserveren. PgBouncer er en slik pooler for PostgreSQL: applikasjonene kobler seg til den som om den var databasen, og den låner dem servertilkoblinger fra sine egne pooler. De to kan kombineres, og Djangos databasereferanse har et avsnitt om å kjøre bak en transaksjonspooler som PgBouncer.

Hvorfor en tilkobling er verdt å beholde, ser man på serversiden. PostgreSQL gir hver klient sin egen prosess: en overordnet prosess, postmaster, lytter etter innkommende tilkoblinger og starter en ny serverprosess for hver tilkoblingsforespørsel, og hver klient er koblet til nøyaktig én av disse prosessene. Å åpne en tilkobling betyr altså å starte en prosess på databaseserveren, og en pool betaler den prisen én gang og gjenbruker så resultatet.

Bruk begrepet i en beslutning

Bruk «Hva er en tilkoblingspool?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Finne spørringene og de manglende indeksene bak et system som har blitt tyngre for hvert år, og rette dem som betyr noe.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. 30 dagers strømdata på 180 ms — 180 ms for å aggregere en måned med måleravlesninger

30 dagers strømdata på 180 ms Ta kontakt →

Hvorfor det er viktig

Den første grunnen er prisen for å koble til. Djangos dokumentasjon sier det rett ut: en vedvarende tilkobling sparer arbeidet med å opprette en ny i hver HTTP-forespørsel. Standardinnstillingen gjør likevel det motsatte. `CONN_MAX_AGE` er 0, som lukker tilkoblingen ved slutten av hver forespørsel; et positivt antall sekunder holder den åpen for gjenbruk, og `None` holder den åpen uten grense. Det er den enkleste formen for gjenbruk, én tilkobling per tråd som lever lenger enn forespørselen, og en pool er den generelle formen.

Den andre grunnen er grensen på serveren. PostgreSQLs `max_connections` er typisk 100 som standard, kan være lavere hvis innstillingene i systemkjernen ikke tåler det, og kan bare endres ved serverstart. Tre av plassene er som standard reservert for superbrukere. Det er heller ikke gratis å heve tallet: PostgreSQL dimensjonerer visse ressurser, blant annet delt minne, direkte etter `max_connections`. Uten en pool holder hver arbeidsprosess sin egen tilkobling, og Djangos dokumentasjon sier hva det fører til: fordi hver tråd har sin egen tilkobling, må databasen kunne ta imot minst like mange samtidige tilkoblinger som det finnes arbeidstråder. En pool gjør det til et tall teamet selv velger og kan summere.

Den tredje grunnen er hygiene. En pool er det ene stedet hver tilkobling passerer mellom to lånere, så det er her en tilkobling blir ryddet og testet. Når en tilkobling kommer tilbake, kaller SQLAlchemys pool `rollback()` på den, som frigjør all transaksjonstilstand, låser medregnet, slik at neste låner ikke arver en halvferdig transaksjon. I PgBouncers standardmodus med sesjonspooling kjøres dens `server_reset_query`, som standard `DISCARD ALL`, når en servertilkobling frigjøres, og før en annen klient får den. Og en pool kan teste en tilkobling før den lånes ut, noe som betyr mye for tilkoblinger som har stått ubrukt en stund.

Slik ser det ut i praksis

I SQLAlchemy 2.1 får hver `Engine` en `QueuePool` som standard, med SQLites database i minnet som unntak. Dens `pool_size` er 5 og dens `max_overflow` er 10, så én engine kan ha opptil 15 tilkoblinger samtidig, og en kaller som finner alle i bruk, venter `pool_timeout`, som standard 30 sekunder, før den gir opp. Innstillingene er nøkkelordargumenter til `create_engine()`:

```
from sqlalchemy import create_engine

# Høyst ti tilkoblinger for denne enginen: ti holdes åpne, ingen utover dem.
engine = create_engine(
    "postgresql+psycopg2://me@localhost/mydb",
    pool_size=10,
    max_overflow=0,
    pool_timeout=10,  # vent ti sekunder på en ledig tilkobling, gi så opp
    pool_pre_ping=True,  # test hver tilkobling når den lånes ut
    pool_recycle=3600,  # bytt ut tilkoblinger som er over en time gamle
)
```

Med `pool_pre_ping` sjekker poolen ved utlån at tilkoblingen fortsatt kan brukes, via driverens egen ping eller en setning som tilsvarer `SELECT 1`. `pool_recycle` er ment for servere som lukker tilkoblinger som har stått ubrukt en viss tid: med 3600 blir enhver tilkobling som har vært åpen i mer enn en time, byttet ut ved neste utlån.

Django 6.1 har sin egen pool for PostgreSQL. Den virker bare med driveren psycopg og krever pakken `psycopg[pool]`, og den slås på med `"OPTIONS": {"pool": True}` i databaseinnstillingene eller med en ordbok med innstillinger for psycopgs `ConnectionPool`. Poolen gjenbruker tilkoblinger på tvers av tråder, og hver prosess har en egen pool for hvert databasealias. `CONN_MAX_AGE` må stå på 0 når pooling er slått på; levetiden til en tilkobling i poolen settes i stedet med poolens innstilling `max_lifetime`. Under ASGI råder Djangos dokumentasjon til å slå av vedvarende tilkoblinger og bruke denne innebygde poolen i stedet.

PgBouncers `pool_mode` avgjør når en servertilkobling kan gå videre til en annen klient. I øktmodus, som er standard, beholder klienten den til den kobler fra. I transaksjonsmodus har en klient bare en servertilkobling mens en transaksjon pågår, så noen dusin servertilkoblinger kan betjene langt flere klienter. Setningsmodus frigjør tilkoblingen etter hver spørring og tillater ikke transaksjoner som strekker seg over flere setninger. Hver pool rommer opptil

`default_pool_size` servertilkoblinger per par av bruker og database, som standard 20, og PgBouncer tar imot opptil `max_client_conn` klienttilkoblinger, som standard 100.

Selskapets egne notater setter poolen tidlig i rekkefølgen. Optimaliser databasen før du sharder den ber leseren telle de aktive øktene i `pg_stat_activity` mot antallet CPU-kjerner: når mange ganger flere er aktive enn det finnes kjerner, bruker serveren tiden sin på å bytte mellom dem, og en tilkoblingspool løser det før noen maskinvare gjør det. Å selge den siste billetten nøyaktig én gang bruker samme tanke på et lynsalg: hundre tusen kjøpere bør bli noen dusin tilkoblinger gjennom PgBouncer, ikke tusenvis av databaseprosesser som venter på én lås.

Hvor det går galt

Poolene krever til sammen flere tilkoblinger enn serveren tillater. Hver pool teller bare sine egne tilkoblinger. Med SQLAlchemys standardverdier kan én engine nå 15, så tolv prosesser kan be om 180 fra en server der `max_connections` typisk er 100 som standard, med tre av dem reservert for superbrukere, og Django har en egen pool for hvert databasealias i hver prosess. Summer det høyeste antallet for hver pool som peker mot databasen, på tvers av alle prosesser i alle applikasjoner og alle planlagte jobber, og hold summen under `max_connections` minus de reserverte plassene. Når det ikke går, sett PgBouncer foran og la den styre antallet servertilkoblinger.

En tilkobling dør mens den venter i poolen. En tilkobling som har stått ubrukt, kan bli lukket i den andre enden uten at poolen vet det, og neste låner feiler da på sin første setning. Slå på `pool_pre_ping` i SQLAlchemy eller `CONN_HEALTH_CHECKS` i Django, som tester en gjenbrukt tilkobling én gang per forespørsel, og sett `pool_recycle` eller psycopgs `max_lifetime` lavere enn tiden før serveren lukker ubrukte tilkoblinger. Ingen av dem dekker en tilkobling som faller midt i en transaksjon: SQLAlchemys dokumentasjon sier klart at pre-ping ikke håndterer det tilfellet, så applikasjonen må selv prøve arbeidet på nytt.

En arbeidsprosess startet med fork arver tilkoblingene til forelderen. En webserver eller en jobbkjører som oppretter enginen sin og deretter starter arbeidsprosessene med fork, etterlater hvert barn med de samme pooltilkoblingene som forelderen. SQLAlchemys dokumentasjon kaller det kritisk at pooltilkoblinger ikke deles med en prosess startet med fork, fordi begge prosessene da ville snakke over den samme nettverkstilkoblingen. Følg den

anbefalte fremgangsmåten og kall `engine.dispose(close=False)` i hvert barns initializer, slik at barnet starter med en tom pool og lar tilkoblingene til forelderen være i fred. Alternativene er `NullPool`, som åpner og lukker en tilkobling ved hver bruk, eller å kalle `engine.dispose()` før fork-kallet.

Transaksjonspooling ødelegger økttilstand. I PgBouncers transaksjonsmodus kan to transaksjoner fra samme klient kjøre på forskjellige servertilkoblinger, så ingenting av det en klient setter opp for økten sin, er garantert å være der i neste transaksjon. PgBouncers funksjonsliste markerer `SET`, `LISTEN`, markører deklarert med `WITH HOLD`, `PREPARE` på SQL-nivå og advisory locks på øktnivå som aldri støttet i den modusen, og legger ikke skjul på at modusen i sin natur bryter noen av antagelsene klienter gjør om serveren. Hold økttilstand ute av kode som går gjennom en transaksjonspooler, og send klientene som trenger den, til en pool i øktmodus. I Django må `DISABLE_SERVER_SIDE_CURSORS` settes til `True` for en database bak en transaksjonspooler, ellers må spørringene kjøre inne i `atomic()`.

Les mer

- Connection Pooling — `QueuePool` og standardverdiene, håndtering av brutte tilkoblinger, nullstilling ved retur og pooler ved fork i SQLAlchemy 2.1.
- PgBouncer features — økt-, transaksjons- og setningsmodus, og hvilke PostgreSQL-funksjoner hver av dem støtter.
- PgBouncer configuration — `pool_mode`, `default_pool_size`, `max_client_conn` og tidsavbruddene med standardverdiene sine.
- How Connections Are Established — PostgreSQLs modell med én prosess per bruker og postmaster-prosessen.
- Connections and Authentication — `max_connections` og tilkoblingsplassene som er reservert for superbrukere og andre roller.
- Databases — vedvarende tilkoblinger, PostgreSQL-poolen og transaksjonspooling i Djangos referanse.

Ordliste

Velg ut fra deres faktiske rammer

Ta med et representativt datasett, spørsmålene det skal besvare og dagens flaskehals. Vi kan avtale målinger av kvalitet, gjennomstrømning og gjenoppretting.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er en tilkoblingspool?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

welcome@engineer.company +45 50 10 61 84 København, Danmark

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/connection-pool/