Engineer

Pool de connexions

réutiliser et limiter les connexions ouvertes d'une base

Ce que c’est

Un pool de connexions est un ensemble de connexions de base de données ouvertes qu’une application conserve et prête à nouveau, au lieu d’en ouvrir une pour chaque opération puis de la fermer. Le code emprunte une connexion au pool, y exécute ses instructions et la rend ; l’appel suivant qui en a besoin reçoit la même connexion, déjà ouverte. La documentation SQLAlchemy décrit ce mécanisme comme « une technique standard utilisée pour conserver en mémoire des connexions de longue durée afin de les réutiliser efficacement ». La même page lui donne un second rôle : gérer le nombre de connexions que l’application peut utiliser simultanément, pour qu’un pic de trafic attende une connexion libre au lieu d’en ouvrir davantage.

Un pool peut se trouver à deux endroits. Dans l’application, il fait partie de la bibliothèque de base de données : chaque `Engine` SQLAlchemy en possède un, et Django peut en garder un dans chaque processus pour PostgreSQL. Hors de l’application, c’est un programme distinct placé entre les clients et le serveur de base de données. PgBouncer remplit ce rôle pour PostgreSQL : les applications s’y connectent comme s’il était la base, et il leur prête des connexions serveur issues de ses propres pools. Les deux approches peuvent se combiner, et la documentation de référence de Django sur les bases de données consacre une section à l’exécution derrière un gestionnaire de pool par transaction tel que PgBouncer.

L’intérêt de garder une connexion apparaît côté serveur. PostgreSQL attribue un processus propre à chaque client : un superviseur appelé postmaster écoute les connexions entrantes, démarre un nouveau processus serveur à chaque demande de connexion, et chaque client se connecte à exactement un de ces processus. Ouvrir une connexion exige donc de démarrer un processus sur le serveur de base de données ; un pool paie ce coût une fois, puis réutilise le résultat.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'un pool de connexions ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Identifier les requêtes et les index manquants dans un système alourdi au fil des ans, puis corriger ce qui compte.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 30 jours de données électriques traités en 180 ms — 180 ms pour agréger un mois de relevés de compteurs

30 jours de données électriques traités en 180 ms Contactez-nous →

Pourquoi c’est important

La première raison est le coût de connexion. La documentation Django expose simplement l’intérêt de conserver les connexions : une connexion persistante évite le surcoût d’en établir une nouvelle à chaque requête HTTP. Pourtant, le comportement préconfiguré fait l’inverse. `CONN_MAX_AGE` vaut 0, ce qui ferme la connexion à la fin de chaque requête ; un nombre positif de secondes la garde ouverte pour réutilisation, et `None` la garde sans limite. C’est la forme la plus simple de réutilisation, une connexion par thread qui survit à la requête ; le pool en est la forme générale.

La deuxième raison est la limite du serveur. `max_connections` de PostgreSQL vaut généralement 100 sans autre réglage, peut être inférieur si les réglages du noyau ne le permettent pas et ne peut changer qu’au démarrage du serveur. Trois de ces emplacements sont réservés aux superutilisateurs sans configuration contraire. Augmenter ce nombre n’est pas gratuit non plus : PostgreSQL dimensionne certaines ressources, dont la mémoire partagée, directement à partir de `max_connections` . Sans pool, chaque worker garde sa connexion, et la documentation Django explicite la conséquence : chaque thread conservant sa propre connexion, la base doit accepter au moins autant de connexions simultanées qu’il existe de threads de travail. Un pool en fait un nombre que l’équipe choisit et peut additionner.

La troisième raison est l’hygiène. Le pool est le point de passage de chaque connexion entre deux emprunteurs : c’est donc là qu’elle est nettoyée et testée. Lorsqu’une connexion revient, le pool SQLAlchemy y appelle `rollback()` , ce qui libère tout état transactionnel, verrous compris, pour que l’emprunteur suivant n’hérite pas d’une transaction à moitié terminée. Dans le mode de regroupement par session utilisé par défaut par PgBouncer, sa `server_reset_query` , `DISCARD ALL` par défaut, est exécutée lorsqu’une connexion serveur est libérée et avant qu’un autre client la reçoive. Un pool peut aussi tester une connexion avant de la prêter, ce qui compte lorsqu’elle est restée inactive un moment.

À quoi cela ressemble en pratique

Dans SQLAlchemy 2.1, chaque `Engine` reçoit un `QueuePool` si aucun autre pool n'est choisi, à l'exception de la base en mémoire de SQLite. Son `pool_size` vaut 5 et son `max_overflow` vaut 10, si bien qu'un moteur peut détenir jusqu'à 15 connexions à la fois. Un appel qui les trouve toutes occupées attend pendant `pool_timeout`, 30 secondes sans autre réglage, avant d'abandonner. Ces réglages sont des arguments nommés de `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
)
```

Avec `pool_pre_ping`, le pool vérifie au moment du prêt que la connexion est encore utilisable, par le ping du pilote ou une instruction équivalente à `SELECT 1`. `pool_recycle` vise les serveurs qui ferment les connexions restées inutilisées un certain temps : avec 3600, toute connexion ouverte depuis plus d'une heure est remplacée au prochain emprunt.

Django 6.1 possède son propre pool pour PostgreSQL. Il ne fonctionne qu'avec le pilote psycopg, exige le paquet `psycopg[pool]` et s'active par `"OPTIONS": {"pool": True}` dans les réglages de base de données, ou par un dictionnaire d'options pour `ConnectionPool` de psycopg. Le pool réutilise les connexions entre threads, et chaque processus garde un pool distinct pour chaque alias de base de données. `CONN_MAX_AGE` doit rester à 0 quand le pool est activé ; la durée de vie d'une connexion du pool se règle alors par son option `max_lifetime`. Sous ASGI, la documentation Django conseille de désactiver les connexions persistantes et d'utiliser plutôt ce pool intégré.

`pool_mode` de PgBouncer détermine quand une connexion serveur peut passer à un autre client. En mode session, le mode préconfiguré, le client la garde jusqu'à sa déconnexion. En mode transaction, il n'a une connexion serveur que pendant l'exécution d'une transaction, si bien que quelques dizaines de connexions

serveur peuvent servir bien plus de clients. Le mode instruction libère la connexion après chaque requête et n'autorise pas les transactions sur plusieurs instructions. Chaque pool contient jusqu'à `default_pool_size` connexions serveur par couple utilisateur et base de données, soit 20 sans autre réglage, et PgBouncer accepte jusqu'à `max_client_conn` connexions client, soit 100 sans autre réglage.

Les notes de l'entreprise placent le pool tôt dans l'ordre des travaux. Optimiser la base avant de la fragmenter invite à compter les sessions actives dans `pg_stat_activity` et à comparer ce nombre à celui des cœurs CPU : lorsqu'elles sont plusieurs fois plus nombreuses que les cœurs, le serveur passe son temps à basculer entre elles, et un pool de connexions résout ce problème avant tout ajout de matériel. Vendre le dernier billet exactement une fois applique la même idée à une vente éclair : cent mille acheteurs doivent devenir quelques dizaines de connexions via PgBouncer, et non des milliers de processus serveur attendant un même verrou.

Où les problèmes surviennent

Le total des pools dépasse la limite du serveur. Chaque pool ne compte que ses connexions. Avec les valeurs prédéfinies de SQLAlchemy, un moteur peut atteindre 15 connexions, si bien que douze processus peuvent en demander 180 à un serveur dont `max_connections` vaut généralement 100 sans autre réglage, avec trois emplacements réservés aux superutilisateurs. Django garde en outre un pool distinct par alias de base de données dans chaque processus. Additionnez le maximum de chaque pool pointant vers la base, dans chaque processus de chaque application et chaque tâche planifiée, et gardez le total sous `max_connections` moins les emplacements réservés. Si cela ne tient pas, placez PgBouncer devant et laissez-le gérer le nombre de connexions serveur.

Une connexion meurt en attendant dans le pool. Une connexion inactive peut être fermée de l'autre côté sans que le pool le sache, et l'emprunteur suivant échoue alors dès sa première instruction. Activez `pool_pre_ping` dans SQLAlchemy ou `CONN_HEALTH_CHECKS` dans Django, qui teste une connexion réutilisée une fois par requête, et réglez `pool_recycle` ou `max_lifetime` de psycopg sous la durée après laquelle le serveur ferme les connexions inactives. Ces mesures ne couvrent pas une connexion interrompue au milieu d'une transaction : la documentation SQLAlchemy dit clairement que pre-ping ne gère pas ce cas, et l'application doit donc réessayer elle-même le travail.

Un worker créé par fork hérite des connexions du parent. Un serveur web ou un exécuteur de tâches qui crée le moteur puis crée ses workers par fork laisse chaque enfant détenir les mêmes connexions du pool que le parent. La documentation SQLAlchemy juge essentiel que les connexions d'un pool ne soient pas partagées avec un processus créé par fork, car les deux processus parleraient alors sur la même connexion réseau. Suivez sa méthode recommandée et appelez `engine.dispose(close=False)` dans l'initialisation de chaque enfant, pour qu'il commence avec un pool vide sans toucher aux connexions du parent ; les autres possibilités sont `NullPool`, qui ouvre et ferme une connexion à chaque usage, ou un appel à `engine.dispose()` avant le fork.

Le pool par transaction casse l'état de session. En mode transaction de PgBouncer, deux transactions du même client peuvent s'exécuter sur des connexions serveur différentes : ce que le client prépare pour sa session n'est donc pas garanti à la transaction suivante. La liste des fonctionnalités de PgBouncer marque `SET`, `LISTEN`, les curseurs déclarés `WITH HOLD`, `PREPARE` au niveau SQL et les verrous consultatifs de session comme jamais pris en charge dans ce mode. Elle reconnaît que, par nature, ce mode rompt certaines hypothèses des clients sur le serveur. Évitez l'état de session dans le code qui passe par un pool par transaction et envoyez les clients qui en ont besoin vers un pool en mode session. Dans Django, réglez `DISABLE_SERVER_SIDE_CURSORS` sur `True` pour une base derrière un pool par transaction, ou exécutez ces requêtes dans `atomic()`.

Pour aller plus loin

- Pooling des connexions — `QueuePool` et ses valeurs prédéfinies, gestion des déconnexions, réinitialisation au retour et pools avec fork dans SQLAlchemy 2.1.
- Fonctionnalités de PgBouncer — modes session, transaction et instruction, et fonctionnalités PostgreSQL prises en charge par chacun.
- Configuration de PgBouncer — `pool_mode`, `default_pool_size`, `max_client_conn` et délais d'attente, avec leurs valeurs prédéfinies.
- Comment les connexions sont établies — le modèle PostgreSQL d'un processus par utilisateur et le postmaster.
- Connexions et authentification — `max_connections` et les emplacements de connexion réservés aux superutilisateurs et aux autres rôles.
- Bases de données — connexions persistantes, pool PostgreSQL et pooling par transaction, dans la référence Django.

Glossaire

Choisissez selon vos contraintes réelles

Présentez un jeu de données représentatif, les questions auxquelles il doit répondre et le goulot d'étranglement actuel. Nous pouvons convenir de la manière de mesurer la qualité, le débit et la reprise.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce qu'un pool de connexions ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/connection-pool/