Engineer

Réplication

Maintenir des copies d'une base à jour sur d'autres serveurs

Ce que c'est

La réplication conserve des copies d'une base de données sur d'autres serveurs et envoie à chacune chaque modification de l'original pour qu'elles restent toutes synchronisées. Lorsque le contexte laisse place au doute, on parle de réplication de base de données. La documentation PostgreSQL appelle primaire le seul serveur qui accepte les écritures, et serveurs de secours ceux qui le suivent. Redis parle de maître et de ses répliques.

PostgreSQL en propose deux types, qui ne copient pas les mêmes choses.

La réplication physique copie le journal d'écriture anticipée, WAL, dans lequel chaque modification est enregistrée avant d'atteindre les fichiers de données. Un serveur de secours rejoue ce journal et devient une copie bloc pour bloc de l'ensemble du cluster. En réplication en continu, le serveur de secours garde une connexion ouverte au primaire et reçoit les enregistrements WAL à mesure de leur création, au lieu d'attendre l'envoi de fichiers WAL terminés. Un serveur de secours qui ne fait que rejouer le journal et n'accepte aucune connexion avant sa promotion est dit tiède, warm standby ; celui qui répond aussi aux requêtes en lecture seule est dit chaud, hot standby.

La réplication logique copie les modifications des lignes. Un éditeur transmet les insertions, mises à jour et suppressions effectuées sur les tables choisies, chaque ligne étant identifiée par son identité de réplication, généralement la clé primaire. Un abonné les applique à ses propres tables. Comme elle traite des lignes plutôt que des blocs de disque, elle peut transporter quelques tables au lieu du cluster entier, même vers une autre version majeure de PostgreSQL.

La réplication en continu est asynchrone par défaut. Le primaire confirme un commit une fois le WAL sur son propre disque et le transmet ensuite. La documentation situe le retard typique à moins d'une seconde lorsque le serveur de secours tient le rythme. Avec la réplication synchrone activée, le commit attend qu'un serveur de secours ait aussi le WAL.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que la réplication ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des sauvegardes testées plutôt que supposées fiables, et un plan écrit pour le matin où un serveur ne redémarre pas.

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. 100 sauvegardes critiques automatisées : Barman — 100 sauvegardes critiques automatisées dans les bases

100 sauvegardes critiques automatisées : Barman Contactez-nous →

Pourquoi c'est important

La première raison est de survivre à la perte d'un serveur. Lorsque le primaire tombe en panne, un serveur de secours qui détient déjà les données peut être promu et reprendre le trafic bien plus vite qu'une sauvegarde ne pourrait être restaurée sur une nouvelle machine.

La deuxième est de répartir les lectures. Dans de nombreuses applications, la plupart des requêtes ne font que lire. Un serveur de secours chaud peut y répondre et laisser le primaire disponible pour les écritures, que lui seul peut accepter.

La troisième est de déplacer les données. La réplication logique peut copier uniquement les tables nécessaires à une base de reporting ou transporter des données vers un serveur utilisant une autre version majeure de PostgreSQL.

Chaque copie impose un choix entre attendre et perdre. La réplication asynchrone ne coûte rien sur le chemin du commit, mais un serveur de secours est toujours légèrement en retard : une lecture peut y manquer une écriture qui vient d'avoir lieu et, si le primaire meurt, des commits déjà confirmés peuvent ne jamais atteindre le serveur qui prend le relais. La réplication synchrone ferme cette fenêtre et ajoute un aller-retour réseau à chaque commit.

La réplication n'est pas non plus une sauvegarde. Un serveur de secours rejoue fidèlement un `DELETE` accidentel sans `WHERE` : en quelques instants, toutes les copies contiennent donc aussi l'erreur. Pour l'annuler, il faut toujours une sauvegarde.

À quoi cela ressemble en pratique

Une boutique exploite sa base sur un primaire à l'adresse `10.0.0.1` et ajoute un second serveur, `db2`, comme réplique en continu. Sur `db2`, avec PostgreSQL installé et arrêté :

```
pg_basebackup -h 10.0.0.1 -U replicator -D /srv/pgdata \
    -R -C -S db2_slot -P
```

`pg_basebackup` copie l'ensemble du cluster en fonctionnement dans le répertoire indiqué par `-D`, qu'il crée si nécessaire et qui doit sinon être vide. Par défaut, il transmet en parallèle le WAL généré pendant la copie. `-C -S db2_slot` crée d'abord un slot de réplication sur le primaire, afin que celui-ci conserve chaque enregistrement WAL jusqu'à sa réception par le nouveau serveur de secours. `-R` crée un fichier `standby.signal` et ajoute les paramètres de connexion, slot compris, à `postgresql.auto.conf`. `-P` indique la progression. Le rôle `replicator` doit avoir l'attribut `REPLICATION` ou les droits superutilisateur, et le `pg_hba.conf` du primaire doit autoriser la connexion de réplication.

Lorsque PostgreSQL démarre sur `db2`, `standby.signal` le fait démarrer comme serveur de secours et recevoir le flux du primaire. Comme `hot_standby` est activé par défaut, il accepte les requêtes en lecture seule une fois son état cohérent. Sur le primaire, une ligne par serveur de secours montre son état :

```
SELECT application_name, state, sync_state, replay_lsn, replay_lag
FROM   pg_stat_replication;
```

Une réplique saine affiche pour `state` la valeur `streaming`, et pour `sync_state` la valeur `async`. `replay_lag` estime le temps qu'il a fallu aux commits récents pour y devenir visibles. Une fois le serveur de secours à jour et l'activité WAL arrêtée, il revient à NULL plutôt qu'à zéro.

La note sur l'optimisation de la base avant de la répartir en shards explique pourquoi une boutique fait cela. La majorité de son trafic correspond à des lectures : page produit, panier, historique des commandes. Rien de tout cela ne nécessite le primaire, qui doit rester disponible pour les écritures. Une réplique en continu prend ces lectures en charge, et la note rappelle qu'en ajouter une relève de la configuration, tandis qu'un shard est un projet. Elle ajoute deux précautions. Une réplique a un léger retard : ce qu'un utilisateur vient d'écrire devrait donc être relu sur le primaire. Et un long rapport sur une réplique peut être annulé s'il retarde les modifications qu'elle rejoue.

Pour qu'un commit attende un serveur de secours, un seul paramètre est nécessaire sur le primaire. Chaque serveur de secours est nommé par l'`application_name` de son `primary_conninfo`, et le primaire énumère les noms qu'il peut attendre :

```
synchronous_standby_names = 'ANY 1 (db2, db3)'
```

Chaque commit attend désormais que son WAL soit sur disque sur le primaire et sur au moins un des serveurs `db2` et `db3`. `synchronous_commit` peut être réglé par utilisateur, par base ou par transaction : l'encaissement d'une commande peut attendre tandis qu'un compteur de pages vues valide à la vitesse locale. La note sur la vente du dernier billet exactement une fois explique l'importance pour le stock. En réplication asynchrone, un primaire peut confirmer une vente et mourir avant que sa réplique la possède. La réplique est promue, la vente a disparu et la même unité est revendue. La solution de la note est un serveur de secours synchrone ou un stockage construit sur Raft, au prix d'une latence supplémentaire à chaque commit.

La réplication logique demande deux instructions. L'éditeur doit avoir `wal_level = logical`, et les tables doivent déjà exister chez l'abonné, car leurs définitions ne sont pas copiées :

```
-- on the publisher
CREATE PUBLICATION shop_pub FOR TABLE orders, customers;

-- on the subscriber
CREATE SUBSCRIPTION shop_sub
    CONNECTION 'host=10.0.0.1 dbname=shop user=replicator'
    PUBLICATION shop_pub;
```

L'abonnement commence par copier les lignes déjà présentes dans ces tables, puis applique chaque modification ultérieure.

Où les problèmes surviennent

Un utilisateur ne voit pas ce qu'il vient d'écrire. L'application écrit sur le primaire puis relit sur une réplique encore légèrement en retard : un paramètre enregistré semble revenir à sa valeur précédente. Orientez les lectures d'un utilisateur vers le primaire pendant un court moment après ses écritures. Si les données doivent être visibles sur le serveur de secours dès le retour du commit, `synchronous_commit = remote_apply` attend que les serveurs de secours synchrones aient rejoué la transaction, avec un coût par commit plus élevé.

Les longues requêtes sur la réplique sont annulées. Le serveur de secours doit rejouer des modifications incompatibles avec une requête en cours : vacuum supprimant des versions de lignes qu'elle peut encore voir, ou verrou exclusif pris par `DROP TABLE` et de nombreuses formes de `ALTER TABLE`. Le rejeu attend jusqu'à `max_standby_streaming_delay`, 30 secondes par défaut, puis annule la requête. Une réplique de reporting peut recevoir un délai plus long et prend alors davantage de retard pendant l'attente. `hot_standby_feedback = on` évite les annulations dues à vacuum, mais la documentation avertit que cela peut faire gonfler le primaire. `pg_stat_database_conflicts` sur le serveur de secours compte les annulations par cause, et l'application devrait réessayer une requête annulée.

Un slot oublié remplit le disque du primaire. Un slot retient le WAL jusqu'à ce que son serveur de secours l'ait reçu, même si ce dernier est éteint ou a disparu depuis longtemps. `max_slot_wal_keep_size` vaut -1 par défaut, ce qui signifie sans limite. Vérifiez régulièrement les slots :

```
SELECT slot_name, active, wal_status, safe_wal_size
FROM   pg_replication_slots;
```

Supprimez le slot d'un serveur de secours disparu. Définir `max_slot_wal_keep_size` plafonne ce qu'un slot peut conserver. Un serveur de secours dont le retard dépasse cette limite trouve son slot invalidé : `wal_status` indique `lost`. `idle_replication_slot_timeout`, désactivé par défaut, invalide les slots restés inactifs plus longtemps que la durée indiquée.

Les écritures s'arrêtent quand un serveur de secours synchrone s'arrête. Avec un seul serveur de secours nommé et celui-ci indisponible, chaque commit sur le primaire attend et peut ne jamais se terminer. Nommez davantage de candidats que le nombre requis, comme dans `ANY 1 (db2, db3)`. À défaut, réduisez ce nombre ou videz `synchronous_standby_names`, puis rechargez la configuration.

Deux serveurs se croient tous deux primaires. PostgreSQL promeut un serveur de secours sur demande, avec `pg_ctl promote` ou `SELECT pg_promote()`, mais ne possède pas de logiciel propre pour détecter la panne d'un primaire, et rien n'empêche l'ancien de revenir et d'accepter les écritures. La documentation avertit que cela mène à « la confusion et, finalement, à la perte de données ». Isolez l'ancien primaire, en l'éteignant ou en coupant son réseau, avant toute promotion, puis reconstruisez-le comme serveur de secours avant son retour.

La réplication logique laisse des éléments derrière elle. Les modifications de schéma ne sont pas répliquées : une colonne ajoutée chez l'éditeur arrête donc la réplication tant que l'abonné ne l'a pas aussi. Appliquez d'abord les changements additifs chez l'abonné. Les séquences ne sont pas non plus répliquées : les lignes arrivent avec leurs identifiants, mais les séquences de l'abonné restent à leurs valeurs de départ et les insertions après un basculement échouent sur des clés dupliquées. Avant de basculer, placez chaque séquence au-delà de la valeur maximale utilisée.

Un maître Redis redémarre vide et entraîne ses répliques. Si la persistance est désactivée sur le maître et qu'un mécanisme le redémarre automatiquement, il revient avec un jeu de données vide que ses répliques copient, supprimant leurs propres données. Gardez la persistance activée ou empêchez le redémarrage automatique du maître.

Pour aller plus loin

- Haute disponibilité — vue d'ensemble PostgreSQL des serveurs primaires et de secours, de la réplication synchrone et asynchrone et du basculement.
- Serveurs de secours par envoi de journaux — réplication en continu, slots de réplication et configuration du mode synchrone.
- Serveur de secours chaud — requêtes en lecture seule sur un serveur de secours, conflits qui les annulent et réglages arbitrant entre retard et annulations.
- pg_basebackup — copie d'un cluster en fonctionnement pour démarrer un serveur de secours, avec toutes les options utilisées ci-dessus.

- Restrictions — ce que la réplication logique ne transporte pas : schémas, séquences, grands objets et tout ce qui n'est pas une table.
- Réplication Redis — suivi d'un maître par ses répliques Redis, caractère asynchrone de la réplication et risque du redémarrage sans persistance.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le parcours de déploiement, les risques opérationnels et les personnes d'astreinte. Nous pouvons définir un changement avec des vérifications, une procédure de reprise et une responsabilité claire.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que la réplication ? » 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/replication/