Étude des mécanismes de distribution dans PostgreSQL

MAC

Samuel Roland et Timothée Van Hove

24 novembre 2024

Table des matières

1. Introduction

Les bases de données relationnelles comme PostgreSQL (PGSQL) ont longtemps dominé le paysage des SGBD, en fournissant des transactions ACID et des requêtes SQL complexes. Pourtant, à mesure que les applications modernes sont devenues plus globales et intensives en données, les limites d'un système sur une seule machine sont devenues évidentes. Que se passe-t-il lorsque cette machine atteint sa capacité maximale ? Que faire si elle tombe en panne ? Et comment gérer des millions d'utilisateurs répartis à travers le monde ?

Ces défis ont ouvert la voie à une explosion des BD NoSQL, conçues dès le départ pour être distribués, sacrifiant souvent la complexité relationnelle pour fournir une disponibilité et scalabilité accrue. Cependant, tout n'est pas une question de « NoSQL contre SQL ». Les bases de données relationnelles, comme PGSQL, ont relevé le défi en évoluant pour intégrer des mécanismes distribués tout en préservant leurs forces historiques.

PGSQL s'est adapté à ces nouvelles exigences grâce à un ensemble d'extensions et de fonctionnalités. Avec des outils comme Citus, qui offre un sharding transparent, ou BDR, qui permet une réplication multi-leader, PGSQL peut rivaliser avec certaines des bases de données distribuées les plus performantes. Ces solutions ne sont pas magiques et impliquent des compromis, mais elles montrent à quel point PGSQL reste pertinent dans ce nouvel écosystème.

2. Réplication

La réplication est au cœur de toute architecture distribuée. Dans PGSQL, elle repose sur la propagation des modifications enregistrées dans les Write-Ahead Logs (WAL) qui permet à PGSQL de synchroniser les répliques avec le leader de manière fiable, bien que les modes de réplication choisis influent sur les performances et la cohérence.

PGSQL offre trois approches principales : la réplication synchrone, asynchrone et logique. Ces mécanismes ont chacun leurs avantages et limites, notamment en termes de latence, de tolérance aux pannes et de cohérence des données.

2.1 La réplication synchrone

Dans la réplication synchrone, PGSQL garantit que toutes les répliques désignées reçoivent et confirment les modifications avant que la transaction ne soit validée. Ce processus repose sur un échange constant de messages entre le leader et les répliques, où chaque réplique doit envoyer un accusé de réception pour signaler qu'elle a bien appliqué les modifications. Ce niveau de coordination renforce la cohérence : une fois qu'une transaction est validée, toutes les répliques synchrones reflètent immédiatement son état.

Cependant, cette approche a un coût. Si une réplique est défaillante ou géographiquement distante, la latence introduite par le réseau peut ralentir considérablement les transactions. Imaginez un leader situé en Europe qui envoie des journaux à une réplique synchronisée en Asie. Même avec des connexions rapides, la latence réseau peut ajouter des dizaines de millisecondes à chaque transaction. Cette attente, bien que supportable pour des systèmes critiques comme ceux des banques, devient problématique dans des applications à haut débit ayant besoin d'une faible latence.

2.2 La réplication asynchrone

La réplication asynchrone privilégie la performance. Contrairement au mode synchrone, le leader n'attend pas que les répliques confirment qu'elles ont reçu les modifications avant de valider une transaction. Ça réduit considérablement la latence, surtout dans des environnements où les répliques sont réparties sur de grandes distances. Par exemple, un utilisateur en Europe peut valider une transaction presque instantanément, même si les répliques en Asie ou en Amérique n'ont pas encore été mises à jour.

Par contre, cette vitesse s'accompagne d'un compromis : le replication lag. Ce décalage peut provoquer des incohérences temporaires, où une réplique asynchrone reflète un état obsolète de la BD. Ça pose des problème dans des scénarios où un utilisateur souhaite lire immédiatement une donnée qu'il vient de mettre à jour, mais où la lecture est dirigée vers une réplique encore en retard (reading your own writes).

2.3 La réplication logique

Pour des cas d'usage plus spécifiques, PGSQL propose la réplication logique. Contrairement à la réplication physique, qui copie les blocs de données, la réplication logique se concentre sur les modifications au niveau des lignes et des transactions. Ça permet de répliquer uniquement certaines tables ou même un sous-ensemble des données, ce qui est utile pour des migrations de bases de données ou l'intégration avec des systèmes tiers.

Par exemple, une entreprise qui migre progressivement ses données vers une nouvelle version de PGSQL peut utiliser la réplication logique pour synchroniser les tables critiques tout en testant la nouvelle configuration. Dans des environnements multi-tenants, où chaque client dispose de sa propre base de données, la réplication logique permet de synchroniser uniquement les données d'un client spécifique, ce qui réduit la surcharge inutile.

2.4 Configuration et mise en œuvre de la réplication

Préparation du leader :

Le serveur principal (leader) doit être configuré pour activer la réplication. Ça inclut les étapes suivantes :

- **Activer le niveau de WAL approprié :** Modifier le paramètre `wal_level` dans `postgresql.conf` et le définir sur `replica` ou `logical` selon le type de réplication souhaité.
- **Configurer le nombre de connexions de répliques :** Le paramètre `max_wal_senders` détermine le nombre maximum de connexions simultanées pour les processus envoyant les journaux WAL. Par défaut, il est fixé à 10.
- **Activer les slots de réplication :** Pour éviter que des journaux nécessaires aux répliques ne soient supprimés, configurer `max_replication_slots` avec un nombre suffisant pour les répliques prévues.

Configuration des répliques :

Les serveurs standby, qu'ils soient pour une réplication synchrone ou asynchrone, nécessitent :

- **Définir `primary_conninfo` :** Fournir les informations de connexion au leader, y compris l'adresse IP ou le nom d'hôte, le port, et les informations d'authentification.
- **Utiliser les slots de réplication :** Associer une réplique à un slot pour garantir la continuité de la réplication, même en cas de déconnexion temporaire.

Paramètres pour la réplication synchrone :

Pour activer la réplication synchrone, configurez :

- `synchronous_standby_names` sur le leader pour spécifier les répliques synchrones. Utiliser `FIRST` ou `ANY` pour prioriser ou définir un quorum de répliques.

Détection des pannes et délai de reprise :

Les paramètres tels que `wal_sender_timeout` et `wal_receiver_timeout` permettent de gérer les déconnexions inattendues entre le leader et les répliques. Par exemple, en cas de panne réseau, ces paramètres déterminent combien de temps attendre avant de marquer une connexion comme perdue.

Surveillance de la réplication :

Des vues comme `pg_stat_replication` permettent de surveiller en temps réel l'état des connexions de réplication, y compris la position du WAL appliquée sur chaque réplique.

3. Cohérence et modèles de consistance

La manière dont on configure notre système de réplication et nos modèles de consistance dépend fortement de nos besoins : cohérence absolue, performances maximales, ou quelque chose entre les deux.

3.1 Cohérence forte

La cohérence forte garantit que toutes les répliques synchrones sont alignées avec le leader dès qu'une transaction est validée. Ce modèle est idéal pour les applications où la moindre divergence est inacceptable, comme les registres financiers. Cependant, il peut entraîner des ralentissements si une ou plusieurs répliques sont indisponibles.

Dans PostgreSQL, la cohérence forte est configurée en activant la réplication synchrone (`synchronous_commit = on`) et en définissant les répliques prioritaires avec `synchronous_standby_names`. Ce niveau de contrôle permet d'équilibrer les performances et la disponibilité.

3.2 Cohérence éventuelle et réplication asynchrone

Avec la réplication asynchrone, la cohérence est éventuelle : les données sur les répliques finissent par refléter l'état du leader, mais avec un retard potentiel. Ce compromis est acceptable pour des cas où la rapidité des transactions est prioritaire, comme les systèmes analytiques ou les réseaux sociaux.

Pour minimiser les effets de ce retard, PostgreSQL offre des outils de surveillance comme `pg_stat_replication`, qui permettent de suivre le décalage entre le leader et ses répliques.

3.3 Résolution des incohérences

Des mécanismes supplémentaires peuvent atténuer les effets des retards dans la réplication asynchrone :

- **Lecture cohérente sur le leader :** Les clients critiques peuvent être configurés pour lire directement sur le leader, où les données sont toujours à jour.
- **Sticky sessions :** Associer un utilisateur à une réplique spécifique garantit des lectures monotones, réduisant les incohérences visibles.

Dans des environnements plus complexes, des outils comme Pgpool-II ou des solutions comme Citus peuvent être utilisés pour router intelligemment les requêtes en fonction des besoins en cohérence.

3.4 Compromis dans PGSQL

La latence réseau est l'un des principaux facteurs limitants. PGSQL utilise un protocole interactif, où chaque commande doit attendre une réponse avant de passer à la suivante. Ça signifie que les transactions avec des répliques distantes, surtout en mode synchrone, peuvent être ralenties par les allers-retours réseau. Même en mode asynchrone, un décalage entre le leader et ses répliques peut entraîner des incohérences qui affectent l'expérience utilisateur.

PGSQL permet de jouer sur plusieurs paramètres pour ajuster la cohérence à nos besoins. On peut opter pour une consistance lecture-écriture stricte en utilisant des transactions serialisables ou en verrouillant explicitement certaines données via `SELECT FOR UPDATE`. Mais **attention** : ces mécanismes augmentent la charge et réduisent la concurrence.

Pour les applications analytiques ou massivement parallèles, une approche moins stricte, combinée avec des vérifications explicites au niveau applicatif, peut suffire. Le tout est de trouver le bon équilibre entre cohérence, performance, et disponibilité.

3.5 Quand la cohérence rencontre la concurrence : MVCC et isolation

Le modèle MVCC de PGSQL garantit que les lectures et les écritures ne se bloquent pas mutuellement, même sous des niveaux d'isolation élevés comme `Serializable`. Ce modèle offre une cohérence instantanée à chaque transaction, tout en minimisant les verrous. Cependant, pour des besoins très stricts, on peut utiliser des transactions serialisables ou des verrous explicites pour éviter les anomalies.

4. Partitionnement et sharding

Le partitionnement et le sharding visent tous deux à découper une base de données, mais leur objectif diffèrent.

Le partitionnement divise une table en plusieurs sous-tables appelées partitions, basées sur un critère défini, comme les dates ou les régions. Tout reste sur un même serveur, ce qui simplifie la gestion et améliore les performances en ciblant uniquement les partitions pertinentes pour une requête.

Le sharding distribue les données entre plusieurs serveurs. Chaque shard est un fragment autonome de la base, stocké sur un nœud distinct. Cette technique est utilisée pour gérer des bases de données trop volumineuses pour un seul serveur ou pour améliorer la disponibilité en répartissant la charge.

Partitionnement et sharding ne s'excluent pas. Par exemple, dans une application de séries temporelles :

- Les données peuvent être partitionnées par plages temporelles (par mois ou trimestre) pour simplifier la gestion des anciennes données et améliorer les performances des requêtes.
- Chaque partition peut ensuite être shardée et distribuée sur plusieurs nœuds grâce à Citus, permettant une scalabilité horizontale pour des charges de travail massives.

PGSQL offre un support natif pour le partitionnement, alors que le sharding distribué repose sur des extensions comme Citus.

4.1 Partitionnement natif dans PGSQL

Depuis la version 10, PGSQL propose un partitionnement natif qui divise une table en sous-tables ou "partitions" selon des critères déclarés au moment de la création de la table. Ça permet d'organiser les données et d'améliorer les performances des requêtes en travaillant sur des ensembles de données plus petits.

Par exemple, pour une table contenant des données de séries temporelles comme des logs , le partitionnement par plages (range partitioning) permet de diviser les données par mois ou année. Quand une requête cible une période donnée, PGSQL ignore automatiquement les partitions non pertinentes grâce à son mécanisme d'exclusion de partitions, ce qui améliore la performance.

PGSQL supporte trois types principaux de partitionnement:

- **Partitionnement par plages (range partitioning)** : Idéal pour les données temporelles ou séquentielles. Exemple : partitionner une table de transactions par mois.
- **Partitionnement par liste (list partitioning)** : Utile pour organiser les données en catégories distinctes comme des régions ou des types d'événements.
- **Partitionnement par hachage (hash partitioning)** : Répartit les données uniformément pour éviter les déséquilibres dans les volumes des partitions.

Les avantages du partitionnement sont multiples :

1. **Amélioration des performances des requêtes** : Les partitions pertinentes sont ciblées, limitant les recherches inutiles.
2. **Suppression efficace des données par sous table** : Les partitions obsolètes peuvent être supprimées rapidement avec `DROP TABLE`, évitant les problèmes de fragmentation. Par ex. toutes les données d'une région peuvent être supprimées facilement dans une seule table si il y a un shard par région.

Cependant, le partitionnement natif reste limité à un nœud unique. Si les données ou la charge augmentent au-delà des capacités d'un serveur, cette approche ne suffit plus.

4.2 Sharding distribué avec Citus : Scalabilité horizontale pour PGSQL

Lorsque les limites d'un seul serveur sont atteintes, le sharding devient une solution incontournable. Contrairement au partitionnement, qui divise les données sur un nœud unique, le sharding distribue les données entre plusieurs nœuds d'un cluster. Ça permet une scalabilité horizontale où chaque nœud gère un sous-ensemble des données, appelé "shard". PGSQL n'implémente pas nativement le sharding, mais des extensions comme Citus transforment PGSQL en une base de données massivement distribuée.

Comment fonctionne Citus ? Les données sont shardées selon une clé de distribution, souvent une colonne comme `user_id` dans une application multi-tenant. Chaque shard est stocké sur un nœud différent, et un nœud coordinateur gère les métadonnées et distribue les requêtes. Par exemple :

- Si une requête concerne un seul utilisateur (par exemple `user_id=123`), elle est routée vers un seul nœud, minimisant les échanges réseau.
- Pour des requêtes plus complexes nécessitant plusieurs shards, Citus parallélise les opérations entre les nœuds, permettant un traitement rapide même sur de gros volumes de données.

Un avantage de Citus est que chaque shard est une table PGSQL normale. Ça veut dire qu'on peut utiliser des index locaux, des contraintes, et d'autres optimisations traditionnelles. De plus, les groupes de shards (shard groups) permettent de regrouper les données fréquemment utilisées ensemble (comme des tables avec des relations fortes), ce qui réduit les opérations entre les nœuds.

4.3 Considérations

Ces techniques nécessitent des choix, notamment pour la clé de distribution dans le sharding. Une mauvaise clé peut entraîner des bottlenecks, car les requêtes devront souvent interroger plusieurs shards.

En plus de ça, le modèle de données doit être fait en prenant en compte des limitations :

- Les tables liées entre elles devraient utiliser la même clé de distribution, pour que les données associées soient regroupées dans le même shard, pour éviter les jointures entre différents nœuds du cluster.
- Les requêtes lourdes (nombreux JOIN, nombreuses entrées) peuvent nécessiter des optimisations pour être parallélisées efficacement.

5. Gestion des pannes et rééquilibrage

Les systèmes distribués sont puissants, mais comme tout ce qui est complexe, ils ne sont pas sans leurs défis. Pannes, déséquilibres, nœuds surchargés : PGSQL, avec ses extensions comme Citus ou Patroni, a des solutions solides pour garder tout ça sous contrôle.

5.1. Détection et récupération des pannes

Quand un nœud tombe, la première étape est de s'en rendre compte. PGSQL ne propose pas directement de mécanisme pour détecter une panne, mais les outils comme Patroni ou Pgpool-II utilisent des heartbeats pour vérifier que tout le monde est en ligne. Si un nœud reste silencieux trop longtemps, il est marqué comme défaillant, et le processus de récupération commence.

Répliques en panne

Quand une réplique plante, PGSQL peut la remettre sur pied grâce à ses journaux WAL. Ces journaux enregistrent toutes les modifications effectuées sur le leader. Lorsqu'une réplique revient à la vie, elle consulte son journal local pour voir où elle s'est arrêtée et demande au leader de lui envoyer les transactions manquantes (catch-up recovery). Ce processus est rapide et garantit que tout reste en ordre.

Le leader en panne

La panne d'un leader est une toute autre histoire. Ici, on doit déclencher un failover : promouvoir une réplique en tant que nouveau leader. Patroni surveille les nœuds via un système comme Etcd ou ZooKeeper et décide, en cas de besoin, qui prendra le relais. Une fois le failover terminé, il faut s'assurer que l'ancien leader ne revienne pas comme si de rien n'était : un mécanisme appelé STONITH (Shoot The Other Node In The Head) empêche un ancien leader défaillant de reprendre son role de leader ce qui créerait une situation avec 2 leaders à la fois. Dans PGSQL, ce mécanisme peut être appliqué à l'aide de `pg_rewind` pour le resynchroniser rapidement avec le nouveau leader. C'est plus rapide que de reconstruire une réplique à partir de zéro.

5.2. Ajouter ou retirer une réplique

Tout commence par une image cohérente de la base de données, prise sur le leader. Cette image est transférée à la nouvelle réplique, qui rattrape ensuite son retard en lisant les journaux WAL. L'opération se déroule sans interruption pour les utilisateurs.

Le retrait d'une réplique est un peu plus compliqué. PGSQL ne redistribue pas automatiquement les données d'un nœud supprimé ; il faut des outils comme Citus pour gérer ça efficacement. Avec Citus, les partitions (shards) gérées par la réplique retirée sont transférées vers d'autres nœuds. Citus s'appuie sur une stratégie de shards fixes, ce qui signifie que seules les partitions nécessaires sont déplacées, limitant les perturbations.

Pour éviter les conflits pendant le retrait d'un shard, Citus introduit les shards orphelins. En gros, le shard n'est pas supprimé immédiatement après son déplacement : il est marqué pour suppression différée, le temps que toutes les requêtes en cours soient terminées. Pas de précipitation, pas de problèmes.

5.3. Rééquilibrage des données

Ajouter ou retirer un nœud change l'équilibre du cluster. Le rééquilibrage consiste à redistribuer les données pour que tout reste fluide. Avec Citus, c'est presque magique.

Quand un nouveau nœud rejoint le cluster, Citus détermine quels shards doivent être déplacés. Le transfert est orchestré de manière transparente, et on peut suivre la progression avec la fonction `get_rebalance_progress`. Elle affiche la taille des shards sur les nœuds source et cible, et estime le pourcentage de transfert terminé.

5.4. Tests réguliers pour une résilience accrue

Un cluster n'est résilient que si ses mécanismes de récupération fonctionnent. Pour éviter les mauvaises surprises, il est crucial de tester régulièrement :

- Les basculements (switchover) pour s'assurer qu'un autre nœud peut prendre le rôle de leader.
- Les simulations de pannes pour vérifier la réactivité des outils comme Patroni.
- Le rééquilibrage automatique pour voir si les données sont bien redistribuées en cas de changement.

6. Conclusion

PGSQL a su s'adapter aux besoins de bases de données distribuées. Avec des outils comme la réplication, le partitionnement et le sharding via des extensions comme Citus, il combine le meilleur des deux mondes : la robustesse des bases relationnelles et la flexibilité des systèmes distribués. Bien sûr, il faut jongler entre cohérence, disponibilité et tolérance aux pannes, comme le rappelle le théorème de CAP. Mais c'est justement là que PGSQL est intéressant, car il offre une palette d'options qui permet d'adapter les solutions aux besoins réels, qu'il s'agisse d'une petite application ou d'un système à grande échelle.

Références

1. Shard Rebalancing in Citus 10.1. Citus Data Blog
2. Citus' Replication Model: Today and Tomorrow Citus Data Blog
3. Cluster Management Citus documentation
4. Failover. PG documentation
5. Replication. PG documentation
6. Concurrency Control PG documentation
7. Understanding partitioning and sharding in Postgres and Citus. Azure Database for PGSQL Blog
8. An Overview of Distributed PGSQL Architectures. Crunchydata Blog
9. How PostreSQL replication works. Medium Blog
10. Database Sharding vs. Partitioning. Baelung Blog