

Streaming replication PSQL

Cette documentation vous guidera à travers le processus de mise en place d'une réplication en streaming entre les instances PSQL. Cette documentation est basée sur le TP de l'UE BDA.

Dans l'objectif d'aborder les problématiques **cloud native**, nous avons choisi d'utiliser **Docker** et **Kubernetes** pour la mise en place de l'infrastructure.

Table des matières

- [Streaming replication PSQL](#)
- [1. Instance creation](#)
- [2. Configuration](#)
 - [2.1. Primary instance](#)
 - [2.2. Secondary instance](#)
- [3. Verification](#)
 - [3.1. Process verification](#)
 - [3.2. Log verification](#)
- [4. Testing replication](#)
 - [4.1. Write and read test](#)
- [5. Monitoring](#)
 - [5.1. From primary instance](#)
 - [5.2. From secondary instance](#)
- [DEBUT DES QUESTIONS:](#)
- [6. Promotion](#)
 - [6.1. Promotion of the secondary instance](#)
 - [6.2. Reconfiguration of the primary instance as secondary](#)
- [7. Configuration as synchronous streaming replication](#)
 - [7.1. Configuration of new secondary instances](#)
 - [7.2. Configuration of synchronous replication](#)
 - [7.3. Tests](#)
- [Bonus : utilisation du QUORUM](#)
- [8. Analyse des slots de réplication](#)
 - [8.1. Analyse des fichiers WAL](#)
- [9. Conclusion](#)

1. Instance creation

Depuis le CLI de docker, exécutez la commande suivante pour entrer dans le conteneur psql **pg0** et **pg1**:

```
docker exec -it <container_name> sh
```

Installation des dépendances, ces packages nous permettront d'utiliser la commande **ps** pour vérifier le processus et **nano** pour éditer les fichiers de configuration.

```
apt update && apt install procps && apt install nano
```

Vous pouvez maintenant créer les instances **pg0** and **pg1** en utilisant la commande suivante:

```
pg_createcluster 15 <instance_name>
```

Pour chaque instance, vous pouvez démarrer et vérifier l'état de l'instance en utilisant les commandes suivantes:

```
pg_ctlcluster 15 <instance_name> start  
pg_ctlcluster 15 <instance_name> status
```

Le résultat devrait être comme ci dessous:

```
pg_ctl: server is running (PID: 100)  
/usr/lib/postgresql/15/bin/postgres "-D" "/var/lib/postgresql/15/pg1" "-c" "config_file=/etc/postgresql/15/pg1/postgresql.conf"
```

2. Configuration

2.1. Primary instance (pg0)

Executer la suite depuis l'utilisateur **postgres**:

```
su - postgres
```

Créer un utilisateur **repluser** avec les privilèges de réplication:

```
$ createuser repluser -p 5433 -P --replication
```

Créer un slot de réplication nommé **pg1_slot** :

Note: la commande ci-dessous retourne le resultat de la fonction **pg_create_physical_replication_slot** qui est **pg1_slot** dans notre cas.

```
$ psql -p 5433 -c "SELECT * FROM  
pg_create_physical_replication_slot('pg1_slot');"
```

Depuis l'instance **pg1** en tant qu'utilisateur postgres, créez un fichier **.pgpass** pour stocker le mot de passe de l'utilisateur **repluser** :

```
$ echo "pg1:5433:replication:repluser:replication" > ~/.pgpass
$ chmod 600 ~/.pgpass
```

Depuis l'instance **pg0** en tant qu'utilisateur postgres, ajoutez les configurations suivantes dans le fichier `/etc/postgresql/15/pg0/postgresql.conf`:

```
listen_addresses = '*'
wal_level = replica
max_wal_senders = 10
wal_sender_timeout = 60s
max_replication_slots = 10
```

Vous pouvez maintenant ajouter la configuration suivante dans le fichier `/etc/postgresql/15/pg0/pg_hba.conf`:

```
host replication repluser 0.0.0.0/0 md5
```

Note: Dans notre cas comme nous utilisons docker, nous avons utilisé `0.0.0.0/0` pour autoriser toutes les adresses IP à se connecter. Dans un environnement de production, il est recommandé de spécifier l'adresse IP de l'instance secondaire.

Vous pouvez maintenant redémarrer l'instance **pg0** pour appliquer les changements:

```
service postgresql restart
```

2.2. Secondary instance

Maintenant que la configuration de l'instance primaire est terminée, nous pouvons passer à la configuration de l'instance secondaire **pg1**.

Note: Comme nous utilisons docker, nous avons utilisé l'adresse IP de l'instance primaire **pg0** pour la configuration de l'instance secondaire **pg1**.

Entant qu'utilisateur **postgres**, vous pouvez maintenant spécifier la configuration du serveur primaire dans le fichier `/etc/postgresql/15/pg1/postgresql.conf`:

```
primary_conninfo = 'host=pg0 port=5433 user=repluser
passfile=/var/lib/postgresql/.pgpass
sslmode=prefer sslcompression=1'
primary_slot_name = 'pg1_slot'
hot_standby = on
wal_receiver_timeout = 60s
```

Vous pouvez maintenant STOPER l'instance secondaire **pg1** pour appliquer les changements:

```
service postgresql stop
```

Supprimez le contenu du répertoire de données de l'instance secondaire **pg1** :

```
$ rm -rf /var/lib/postgresql/15/pg1/*  
su - postgres
```

Puis appliquez la sauvegarde de l'instance primaire **pg0** sur l'instance secondaire **pg1** :

```
$ pg_basebackup -h pg0 -p 5433 -D /var/lib/postgresql/15/pg1/ -U repluser -  
v -P -X stream -c fast
```

Sortie de la commande:

```
pg_basebackup: write-ahead log end point: 0/2000100  
pg_basebackup: waiting for background process to finish streaming ...  
pg_basebackup: syncing data to disk ...  
pg_basebackup: renaming backup_manifest.tmp to backup_manifest  
pg_basebackup: base backup completed
```

Configurer l'instance **pg1** comme instance secondaire, mode **standby**, puis redémarrer l'instance:

```
$ touch /var/lib/postgresql/15/pg1/standby.signal  
$ service postgresql restart
```

3. Verification

3.1. Process verification

Pour contrôler si les processus **walreceiver** et **walsender** sont en cours d'exécution, vous pouvez utiliser la commande suivante:

```
ps aux | grep -E 'walsender | walreceiver'
```

Depuis l'instance **pg0** vous devriez voir le processus **walsender** en cours d'exécution:

```
postgres      480  0.0  0.0 218532 12564 ?        Ss   17:25   0:00  
postgres: 15/pg0: walsender repluser 172.26.0.3(41166) streaming 0/3000148
```

Depuis l'instance **pg1** vous devriez voir le processus **walreceiver** en cours d'exécution:

```
postgres      437  0.0  0.0 217352 13716 ?          Ss   17:25   0:00
postgres: 15/pg1: walreceiver streaming 0/3000148
```

3.2. Log verification

Vous pouvez aussi vérifier les logs pour voir si la réplication fonctionne correctement. Depuis l'instance **pg1**, vous pouvez utiliser la commande suivante pour vérifier le log:

```
tail -n 100 /var/log/postgresql/postgresql-15-pg1.log
```

La sortie devrait ressembler à ceci:

```
2024-02-06 17:25:24.362 UTC [437] LOG:  started streaming WAL from primary at 0/3000000 on timeline 1
2024-02-06 17:30:24.385 UTC [434] LOG:  restartpoint starting: time
2024-02-06 17:30:24.427 UTC [434] LOG:  restartpoint complete: wrote 1 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1 recycled; write=0.005 s, sync=0.001 s, total=0.042 s; sync files=0, longest=0.000 s, average=0.000 s; distance=16384 kB, estimate=16384 kB
2024-02-06 17:30:24.427 UTC [434] LOG:  recovery restart point at 0/3000060
```

4. Testing replication

Pour tester la réplication, vous pouvez créer une table à partir de l'instance primaire **pg0** et vérifier si la table est créée dans l'instance secondaire **pg1**.

Créez une base de données **music** à partir de l'instance primaire **pg0** (voir les données du TP)

```
$ psql -p 5433 -c "create database music;"
```

Depuis l'instance secondaire **pg1**, vous pouvez vérifier si la base de données **music** a bien été créée:

```
$ psql -p 5433 -d music -c "\dt"
```

La sortie devrait ressembler à ceci:

| List of databases | | | | | | | |
|-------------------|----------|-----------|---------|-------|------------|-----------------|-----------------------|
| Name | Owner | Encoding | Collate | Ctype | ICU Locale | Locale Provider | Access privileges |
| music | postgres | SQL_ASCII | C | C | | libc | |
| postgres | postgres | SQL_ASCII | C | C | | libc | |
| template0 | postgres | SQL_ASCII | C | C | | libc | =c/postgres + |
| | | | | | | | postgres=CtC/postgres |
| template1 | postgres | SQL_ASCII | C | C | | libc | =c/postgres + |
| | | | | | | | postgres=CtC/postgres |

(4 rows)

4.1. Write and read test

Pour être sûr qu'il est impossible d'écrire sur l'instance secondaire **pg1**, vous pouvez essayer de créer une table dans la base de données **music** depuis celle-ci:

```
psql -p 5433 -c "create database write;"
```

La sortie devrait ressembler à ceci:

```
ERROR: cannot execute CREATE DATABASE in a read-only transaction
```

Essayons maintenant de lire les données de la table `artists` depuis l'instance secondaire `pg1`:

```
psql -p 5433 -d music -c "select * from artists;"
```

La sortie devrait ressembler à ceci:

```
postgres@916e090fb3c3:~$ psql -p 5433 -d music -c "SELECT * FROM \"albums\";"
albumid |          name          | format | year | online_stores | artistid
-----+-----+-----+-----+-----+-----
      1 | Tribute to Stephane Grappelli | CD      | 2000 | fnac           |      3
      2 | Constellation           | digital | 2018 | bandcamp       |      2
      3 | Kaleidoscope            | digital | 2021 | bandcamp       |      2
      4 | You must believe in spring | digital vinyl | 2004 | fnac           |      1
      5 | Live in Paris           | CD      | 2002 | amazon         |      4
(5 rows)
```

5. Monitoring

5.1. From primary instance

La réplication peut être monitoré à partir de l'instance primaire en utilisant la commande suivante:

```
$ psql -p 5433 -x -c "select * from pg_stat_replication;"
```

5.2. From secondary instance

Depuis l'instance secondaire, les métriques de réplication peuvent être surveillées en utilisant la commande suivante:

```
$ psql -p 5433 -x -c "select * from pg_stat_wal_receiver;"
```

DEBUT DES QUESTIONS:

Comparez les informations sur la réplication avec celles observées sur le primaire. Que remarquez-vous ?

Ce que nous remarquons c'est que les informations sur la réplication concordent avec le primaire et le secondaire. On remarque que la dernière version du WAL envoyé est la même que celle reçue.

Affichez les métriques de réplication du serveur primaire et secondaire. Observez les informations de réplication. Que remarquez-vous ? Que pouvez-vous en déduire ?

Ce que nous remarquons c'est la version du WAL a changé et a bien été mise à jour et reçue par le secondaire. Nous remarquons aussi que la propriété `last_msg_send_time` a bien été mise à jour avec la dernière la dernière insertion effectuée.

| PRIMARY | |
|------------------|-------------------------------|
| pid | 554 |
| usesysid | 16388 |
| username | repluser |
| application_name | 15/pg1 |
| client_addr | 192.168.160.3 |
| client_hostname | |
| client_port | 48268 |
| backend_start | 2024-02-06 21:35:03.678864+00 |
| backend_xmin | |
| state | streaming |
| sent_lsn | 0/344A890 |
| write_lsn | 0/344A890 |
| flush_lsn | 0/344A890 |
| replay_lsn | 0/344A890 |

| SECONDARY | |
|-----------------------|-------------------------------|
| pid | 546 |
| status | streaming |
| receive_start_lsn | 0/3000000 |
| receive_start_tli | 1 |
| written_lsn | 0/344A890 |
| flushed_lsn | 0/344A890 |
| received_tli | 1 |
| last_msg_send_time | 2024-02-06 21:45:45.895503+00 |
| last_msg_receipt_time | 2024-02-06 21:45:45.895539+00 |
| latest_end_lsn | 0/344A890 |
| latest_end_time | 2024-02-06 21:39:45.095646+00 |
| slot_name | pg1_slot |
| sender_host | pg0 |
| sender_port | 5433 |

| PRIMARY | |
|------------------|-------------------------------|
| pid | 554 |
| usesysid | 16388 |
| username | repluser |
| application_name | 15/pg1 |
| client_addr | 192.168.160.3 |
| client_hostname | |
| client_port | 48268 |
| backend_start | 2024-02-06 21:35:03.678864+00 |
| backend_xmin | |
| state | streaming |
| sent_lsn | 0/71DF760 |
| write_lsn | 0/71DF760 |
| flush_lsn | 0/71DF760 |
| replay_lsn | 0/71DF760 |

| SECONDARY | |
|-----------------------|-------------------------------|
| pid | 546 |
| status | streaming |
| receive_start_lsn | 0/3000000 |
| receive_start_tli | 1 |
| written_lsn | 0/71DF760 |
| flushed_lsn | 0/71DF760 |
| received_tli | 1 |
| last_msg_send_time | 2024-02-06 22:01:42.122284+00 |
| last_msg_receipt_time | 2024-02-06 22:01:42.122312+00 |
| latest_end_lsn | 0/71DF760 |
| latest_end_time | 2024-02-06 22:00:41.983325+00 |
| slot_name | pg1_slot |
| sender_host | pg0 |
| sender_port | 5433 |

Calculez la quantité de données transmises entre les instances `pg0` et `pg1` depuis le début de la réplication.

Pour calculer la quantité de données transmises entre les instances `pg0` et `pg1` depuis le début de la réplication, nous pouvons utiliser la requête suivante:

```
psql -c "select pg_wal_lsn_diff('0/71DF760', '0/3000000');"
```

Total data transmis: **69072736 octets**

Listez les lots de réplication et observer les valeurs « active » et « restart_lsn » pour le slot `pg1_slot`. Que remarquez-vous ?

On remarque que le slot `pg1_slot` est actif avec une valeur `t` (qui signifie true) et que la valeur `restart_lsn` correspond à la dernière position du WAL envoyé qui est la version qui sera utilisée en cas de redémarrage.

On remarque aussi que le "sender" correspond bien à la valeur `pg0` qui est le serveur primaire.

6. Promotion

6.1. Promotion of the secondary instance

Pour promouvoir l'instance secondaire **pg1** en tant que serveur primaire, vous pouvez utiliser la commande suivante:

```
$ pg_ctlcluster 15 pg1 promote
```

La requête abouti-t-elle ? Qu'en déduisez-vous ? Affichez les logs de l'instance **pg1** (dans `/var/log/postgresql/postgresql-15-pg1.log`) et montrez que la promotion a eu lieu. Vérifiez également l'emplacement dans le WAL. Tentez de créer une nouvelle table dans la base « **music** » depuis l'instance **pg1** avec la requête suivante :

On remarque que la requête abouti et que le serveur secondaire a bien été promu en serveur primaire. On note aussi qu'il est maintenant possible d'écrire sur le serveur **pg1** qui a été promu en serveur primaire.

```
2024-02-06 22:58:28.606 UTC [758] LOG:  received promote request
2024-02-06 22:58:28.633 UTC [758] LOG:  selected new timeline for recovery
2024-02-06 22:58:28.703 UTC [758] LOG:  archive recovery complete
2024-02-06 22:58:28.730 UTC [755] LOG:  database system is ready to accept connections
2024-02-06 22:58:28.761 UTC [756] LOG:  checkpoint complete: wrote 2 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.008 s,
test=0.002 s, average=0.002 s; distance=0 kB, estimate=0 kB
postgres@111c0d168c75:~$ psql -p 5433 -x -c "select * from pg_stat_replication;"
(0 rows)
postgres@111c0d168c75:~$ psql -p 5433 -d music -c "create table testwrite(id integer);"
CREATE TABLE
```

6.2. Reconfiguration of the primary instance as secondary

Affichez les logs de l'instance **pg0** (`/var/log/postgresql/postgresql-15-pg0.log`) et montrez qu'elle est bien passé en instance secondaire (ou « **standby** »).

Depuis les logs de l'instance **pg0**, on remarque que le serveur a bien été reconfiguré en serveur secondaire : **entering standby mode**. De plus, on remarque également en listant les processus que le processus **walsender** a bien été arrêté pour laisser place au processus **walreceiver**.

```
2024-02-07 14:11:59.575 UTC [2307] LOG:  database system was interrupted; last known up at 2024-02-07 14:07:33 UTC
2024-02-07 14:11:59.637 UTC [2307] LOG:  entering standby mode
2024-02-07 14:11:59.640 UTC [2307] LOG:  redo starts at 0/60000020
2024-02-07 14:11:59.650 UTC [2307] LOG:  consistent recovery state reached at 0/6000100
2024-02-07 14:11:59.650 UTC [2304] LOG:  database system is ready to accept read-only connections
2024-02-07 14:11:59.663 UTC [2308] LOG:  started streaming WAL from primary at 0/7000000 on timeline 2
postgres 2308 0.0 0.0 220368 14108 ? Ss 14:11 0:02 postgres: 15/pg0: walreceiver streaming 0/7000148
```

Analysez les métriques de réplication sur **pg1** et **pg0**. Les deux instances se trouvent-elles au même emplacement dans le WAL ?

On note que les deux instances se trouvent au même emplacement dans le WAL. On remarque que la dernière version du WAL envoyé est la même que celle reçue. On en déduit que les deux instances sont bien synchronisées.


```

pid          | 2308
status       | streaming
receive_start_lsn | 0/7000000
receive_start_tli | 2
written_lsn   | 0/7000148
flushed_lsn   | 0/7000148
received_tli  | 2
last_msg_send_time | 2024-02-07 15:12:49.911641+00
last_msg_receipt_time | 2024-02-07 15:12:49.911705+00
latest_end_lsn | 0/7000148
latest_end_time | 2024-02-07 14:12:41.272586+00
slot_name    | pg0_slot
sender_host   | pg1
sender_port   | 5433

```

SECONDARY
pg0

```

pid          | 1328
usesysid     | 16384
username     | repluser
application_name | 15/pg0
client_addr  | 192.168.160.2
client_hostname | 
client_port  | 51678
backend_start | 2024-02-07 14:11:59.656966+00
backend_xmin  | 
state        | streaming
sent_lsn     | 0/7000148
write_lsn    | 0/7000148
flush_lsn    | 0/7000148
replay_lsn   | 0/7000148

```

PRIMARY
pg1

Après avoir créé une nouvelle table dans la base `music` depuis l'instance `pg1`, on remarque que la table a bien été créée et synchronisée sur l'instance `pg0`.

```

List of relations
Schema | Name   | Type  | Owner
-----+-----+-----+-----
public | albums | table | postgres
public | artists | table | postgres
public | toto   | table | postgres
(3 rows)

```

7. Configuration as synchronous streaming replication

7.1. Configuration of new secondary instances

Montrez que les deux nouveaux secondaires sont bien reliés au primaire.

Depuis la commande :

```

psql -p 5433 -x -c "select application_name, state, sent_lsn, write_lsn,
flush_lsn, replay_lsn, sync_state from pg_stat_replication;"

```

On remarque que les deux nouveaux secondaires sont bien reliés au primaire. On remarque aussi qu'ils sont tous en mode `async`. C'est à dire que chaque transaction est répliquée de manière asynchrone.

```

-[ RECORD 1 ]-----+-----
application_name | 15/pg0
state            | streaming
sent_lsn         | 0/11000060
write_lsn        | 0/11000060
flush_lsn        | 0/11000060
replay_lsn       | 0/11000060
sync_state       | async
-[ RECORD 2 ]-----+-----
application_name | 15/pg2
state            | streaming
sent_lsn         | 0/11000060
write_lsn        | 0/11000060
flush_lsn        | 0/11000060
replay_lsn       | 0/11000060
sync_state       | async
-[ RECORD 3 ]-----+-----
application_name | 15/pg3
state            | streaming
sent_lsn         | 0/11000060
write_lsn        | 0/11000060
flush_lsn        | 0/11000060
replay_lsn       | 0/11000060
sync_state       | async

```

7.2. Configuration of synchronous replication

Montrez que l'instance sync2 est devenue une instance synchrone

Après avoir configuré l'instance **pg2** en tant que serveur synchrone, on remarque que l'instance **pg2** est bien devenue une instance synchrone.

```

-[ RECORD 3 ]-----+-----
application_name | sync2
state            | streaming
sent_lsn         | 0/110001F8
write_lsn        | 0/110001F8
flush_lsn        | 0/110001F8
replay_lsn       | 0/110001F8
sync_state       | sync

```

Essayez de créer une nouvelle table dans la base de données **music** avec la requête ci-dessous. Que se passe-t-il ?

Après avoir essayé de créer une nouvelle table dans la base de données **music** depuis l'instance **pg1**, on remarque que la requête abouti et que la table a bien été créée.

Essayez de faire la même chose, mais cette fois-ci en arrêtant au préalable l'instance **pg2**. Que se passe-t-il ?

Après avoir arrêté l'instance **pg2** et essayé de créer une nouvelle/supprimer la table dans la base de données **music** depuis l'instance **pg1**, on remarque que la requête n'aboutit pas directement mais que la transaction a été committée seulement localement (après avoir annulé la transaction manuellement `ctrl+c`). On peut supposer qu'une requête client dans un environnement de production aurait été bloquée jusqu'à ce que l'instance **pg2** soit redémarrée.

```
postgres@90df98b88b94:~$ psql -p 5433 -d music -c "DROP table sync;"
^C
Cancel request sent
WARNING: canceling wait for synchronous replication due to user request
DETAIL: The transaction has already committed locally, but might not have been replicated to the standby.
DROP TABLE
```

Une fois l'instance **pg2** redémarrée, on remarque que la transaction a bien été répliquée sur l'instance **pg2**. On peut justifier cela en testant la commande suivante depuis l'instance **pg2** après avoir redémarré :

```
postgres@d0355afab538:~$ psql -p 5433 -d music -c "select * from sync;"
 id | backend_start | state | sync_state | FROM
----+-----+-----+-----+----
(0 rows)
```

Listez les instances connectées au serveur primaire. Quel est l'état de **sync2** ? Quel est l'état de **sync3** ?

Après avoir listé les instances connectées au serveur primaire, on remarque que l'instance **pg0** est bien en mode **sync** et que l'instance **pg3** quant à elle est en mode **potential**. Cela signifie que l'instance **pg3** est en attente de devenir une instance synchrone (instance de secours). La propriété **FIRST 1** définit la première instance déclarée comme étant obligatoire pour la synchronisation, rendant ainsi l'instance **pg3** dans un état **potential**.

```
-[ RECORD 2 ]-----+-----
application_name | sync3
state             | streaming
sent_lsn          | 0/1105C480
write_lsn         | 0/1105C480
flush_lsn         | 0/1105C480
replay_lsn        | 0/1105C480
sync_state        | potential
-[ RECORD 3 ]-----+-----
application_name | 15/pg0
state            | streaming
sent_lsn         | 0/1105C480
write_lsn        | 0/1105C480
flush_lsn        | 0/1105C480
replay_lsn       | 0/1105C480
sync_state        | async
```

Créez une nouvelle table dans la base de données **music** avec la commande ci-dessous, La requête aboutit-elle ? La nouvelle table est-elle présente sur **sync3** ?

La requête aboutit et la table est bien présente sur l'instance **pg3**.

Arrêtez sync3 et essayez de créer une nouvelle table dans la base music. L'arrêt du potentiel empêche-t-il l'exécution des requêtes ?

On peut constater que la requête s'exécute normalement sans erreur ou warning d'exécution.

Redémarrez sync3 et arrêtez sync2. Que remarquez-vous sur l'état de sync3 ? Qu'en déduisez-vous ?

On remarque que l'instance **pg3** est bien devenue une instance synchrone. On en déduit donc que l'arrêt de l'instance **pg2** a permis à l'instance **pg3** de devenir une instance synchrone de remplacement.

```

-[ RECORD 1 ]-----+-----
application_name | sync3
state            | streaming
sent_lsn         | 0/1108B0A0
write_lsn        | 0/1108B0A0
flush_lsn        | 0/1108B0A0
replay_lsn       | 0/1108B0A0
sync_state       | sync

```

Dans `/etc/postgresql/15/pg1/postgresql.conf`, passez à deux synchrones simultanés en passant la valeur 1 à 2 à la ligne `synchronous_standby_names` comme ci-dessous :

`synchronous_standby_names = 'FIRST 2 (sync2, sync3)'` Redémarrez l'instance **pg1** pour appliquer les changements. Quel est maintenant l'état de sync3 ?

On remarque que l'instance **pg3** est bien devenue une instance synchrone, cela s'explique par la propriété **FIRST 2** ajouté précédemment qui définit les deux premières instances déclarées comme étant obligatoires pour la synchronisation.

7.3. Tests

Arrêtez sync3 (**pg3**). Que se passe-t-il si vous essayez de créer une nouvelle table dans la base de données music ? Redémarrez ensuite sync3. La requête aboutit-elle ? Que pouvez-vous en déduire ?

Tout comme la question précédente, on remarque que la requête s'exécute mais retourne cependant des **warnings** d'exécution.

```

psql -p 5433 -d music -c "create table synchronTest(id integer);"
WARNING: canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have
been replicated to the standby.
CREATE TABLE

```

Après avoir redémarré l'instance **pg3**, on remarque que la transaction a bien été répliquée .

```

postgres@cf932b694ba1:~$ psql -p 5433 -d music -c "select * from
synchronTest;"
 id

```

```
----
(0 rows)
```

Bonus : utilisation du QUORUM

Observez l'état des serveurs et montrez que les trois instances secondaires font désormais partie du quorum.

On remarque que les trois instances secondaires font désormais partie du quorum. Le mot-clé **ANY**, utilisé avec **synchronous_standby_names**, spécifie une réplication synchrone basée sur un quorum, si bien que chaque validation de transaction attendra jusqu'à ce que les enregistrements des WAL soient répliqués de manière synchrone sur au moins **synchronous_standby_names** des serveurs secondaires listés.

```
-[ RECORD 1 ]-----+-----
application_name | sync0
state            | streaming
sent_lsn         | 0/110A19A8
write_lsn        | 0/110A19A8
flush_lsn        | 0/110A19A8
replay_lsn       | 0/110A19A8
sync_state       | quorum
-[ RECORD 2 ]-----+-----
application_name | sync3
state            | streaming
sent_lsn         | 0/110A19A8
write_lsn        | 0/110A19A8
flush_lsn        | 0/110A19A8
replay_lsn       | 0/110A19A8
sync_state       | quorum
-[ RECORD 3 ]-----+-----
application_name | sync2
state            | streaming
sent_lsn         | 0/110A19A8
write_lsn        | 0/110A19A8
flush_lsn        | 0/110A19A8
replay_lsn       | 0/110A19A8
sync_state       | quorum
```

Arrêtez sync3 et essayez de créer de nouveau une table dans la base music. Que se passe-t-il ? Que pouvez-vous en déduire sur le rôle du quorum ?

On remarque que la requête s'exécute normalement sans erreur ou warning d'exécution. On peut en déduire que le quorum permet de garantir la disponibilité des données en cas de panne d'une instance secondaire, synchrone.

```
postgres@90df98b88b94:~$ psql -p 5433 -d music -c "create table quorum(id
integer);"
CREATE TABLE
```

8. Analyse des slots de réplication

8.1. Analyse des fichiers WAL

Listez le contenu de `/var/lib/postgresql/15/pg1/pg_wal`

```
postgres@90df98b88b94:/$ ls -lah /var/lib/postgresql/15/pg1/pg_wal/
total 81M
drwx----- 3 postgres postgres 4.0K Feb  7 16:17 .
drwx----- 19 postgres postgres 4.0K Feb  8 11:11 ..
-rw----- 1 postgres postgres  41 Feb  7 13:51 00000002.history
-rw----- 1 postgres postgres 16M Feb  8 11:46 00000002000000000000000000000011
-rw----- 1 postgres postgres 16M Feb  7 16:08 00000002000000000000000000000012
-rw----- 1 postgres postgres 16M Feb  7 16:08 00000002000000000000000000000013
-rw----- 1 postgres postgres 16M Feb  7 16:12 00000002000000000000000000000014
-rw----- 1 postgres postgres 16M Feb  7 16:12 00000002000000000000000000000015
drwx----- 2 postgres postgres 4.0K Feb  7 08:16 archive_status
```

Listez de nouveau le contenu du dossier `pg_wal` et comparez le résultat à celui de la question. Que pouvez-vous observer comme différences ? (regardez notamment la date et l'heure de dernière modification des fichiers)? Que pouvez-vous déduire des résultats observés ci-dessus ?

Après avoir listé de nouveau le contenu du dossier `pg_wal`, on remarque que la date et l'heure de dernière modification des fichiers ont changé : **Feb 7 16:08** avant insertion et **Feb 8 11:57** après insertion. Ainsi, on peut en déduire que les fichiers ont été correctement mis à jour sur au moins 2 des 3 instances secondaires.

```
postgres@90df98b88b94:/$ ls -lah /var/lib/postgresql/15/pg1/pg_wal/
total 81M
drwx----- 3 postgres postgres 4.0K Feb  7 16:17 .
drwx----- 19 postgres postgres 4.0K Feb  8 11:11 ..
-rw----- 1 postgres postgres  41 Feb  7 13:51 00000002.history
-rw----- 1 postgres postgres 16M Feb  8 11:57 00000002000000000000000000000011
-rw----- 1 postgres postgres 16M Feb  8 11:57 00000002000000000000000000000012
-rw----- 1 postgres postgres 16M Feb  8 11:57 00000002000000000000000000000013
-rw----- 1 postgres postgres 16M Feb  8 11:57 00000002000000000000000000000014
-rw----- 1 postgres postgres 16M Feb  8 11:57 00000002000000000000000000000015
drwx----- 2 postgres postgres 4.0K Feb  7 08:16 archive_status
```

Depuis l'instance secondaire `pg2`, on peut vérifier que les fichiers ont bien été répliqués en cherchant si la base de données `testwal` est bien présente.

```
postgres@d0355afab538:~$ psql -p 5433 -d testwal -c "\dt"
List of relations
Schema | Name | Type  | Owner
-----+-----+-----+-----
public | t1   | table | postgres
(1 row)
```

Après avoir lancé la commande : `psql -p 5434 -c 'CHECKPOINT;'`

Listez de nouveau le contenu du dossier `pg_wal`. Quels changements pouvez-vous observer par rapport aux résultats précédents ? Que pouvez-vous déduire de ces observations ?

Après le "CHECKPOINT" on remarque que la valeur hexadécimale des fichiers composant le WAL a changé. On peut en déduire que la fonction **CHECKPOINT** a été correctement exécutée.

Il est important de noter que la fonction **CHECKPOINT** force une écriture des données sur le disque sans attendre le CHECKPOINT régulier planifié par le système. De plus il force la réécriture des fichiers WAL, comme fichiers de démarrage pour les futures transactions. Cette commande ne s'utilise généralement pas dans un environnement de production.

```
drwx----- 3 postgres postgres 4.0K Feb  8 12:27 .
drwx----- 19 postgres postgres 4.0K Feb  8 12:21 ..
-rw----- 1 postgres postgres  41 Feb  7 13:51 00000002.history
-rw----- 1 postgres postgres 16M Feb  8 12:27 00000002000000000000000019
-rw----- 1 postgres postgres 16M Feb  8 12:26 0000000200000000000000001A
-rw----- 1 postgres postgres 16M Feb  8 12:26 0000000200000000000000001B
-rw----- 1 postgres postgres 16M Feb  8 12:26 0000000200000000000000001C
-rw----- 1 postgres postgres 16M Feb  8 12:26 0000000200000000000000001D
drwx----- 2 postgres postgres 4.0K Feb  7 08:16 archive_status
```

Arrêtez l'instance pg2, créez une nouvelle table dans la base « testwal » et y insérer des données. Observez l'état des fichiers WAL et montrez que les données y ont bien été écrites.

Après avoir arrêté l'instance **pg2** et créé une nouvelle table dans la base **testwal** depuis l'instance **pg1**, on remarque que l'heure de dernière modification des fichiers WAL a bien changé. On peut en déduire que les données ont bien été répliquées.

```
postgres@90df98b88b94:~$ ls -lah /var/lib/postgresql/15/pg1/pg_wal/
total 81M
drwx----- 3 postgres postgres 4.0K Feb  8 12:27 .
drwx----- 19 postgres postgres 4.0K Feb  8 12:21 ..
-rw----- 1 postgres postgres  41 Feb  7 13:51 00000002.history
-rw----- 1 postgres postgres 16M Feb  8 12:38 00000002000000000000000019
-rw----- 1 postgres postgres 16M Feb  8 12:38 0000000200000000000000001A
-rw----- 1 postgres postgres 16M Feb  8 12:38 0000000200000000000000001B
-rw----- 1 postgres postgres 16M Feb  8 12:38 0000000200000000000000001C
-rw----- 1 postgres postgres 16M Feb  8 12:38 0000000200000000000000001D
drwx----- 2 postgres postgres 4.0K Feb  7 08:16 archive_status
```

Observez l'état des fichiers WAL. Les fichiers ont-ils été recyclés ? Pourquoi ?

Nous constatons que les fichiers n'ont pas été recyclés. Cela s'explique par le fait que l'instance **pg2** est en mode **async** et que les données n'ont pas encore été répliquées sur l'instance **pg2**, bloquant ainsi le recyclage des fichiers WAL, les conservant en attente dans le slot de réplication.

Redémarrez pg2 et vérifiez qu'il est bien synchronisé avec le primaire. Vérifiez pour cela les champs LSN sur le primaire et pg2.

Après avoir redémarré l'instance **pg2**, on remarque que l'instance celle-ci est bien synchronisée avec le primaire. On peut le justifier en comparant les champs LSN sur le primaire **pg1** et le secondaire **pg2** à l'aide des commandes suivantes:

Depuis le secondaire:

```
psql -p 5433 -x -c "select * from pg_stat_wal_receiver;
```

Depuis le primaire:

```
psql -p 5433 -x -c "select * from pg_stat_replication;"
```

Ensuite nous pouvons relancer la commande `psql -p 5434 -c 'CHECKPOINT;'` pour vérifier que les fichiers WAL ont bien été recyclés cette fois-ci.

Observez l'état des fichiers WAL. Les fichiers ont-ils été recyclés ? Pourquoi ?

On remarque que la valeur hexadécimale des fichiers WAL a bien changé. On peut en déduire que les fichiers ont bien été réécrits. Cela s'explique par le fait que l'instance `pg2` est bien synchronisée avec le primaire et que les données ont bien été répliquées.

9. Conclusion

En conclusion de ce TP, la mise en place réussie d'une réplication en streaming entre les instances PSQL, avec la configuration des serveurs primaire et secondaire (pg0 et pg1), ainsi que des serveurs secondaires synchrone puis asynchrone, a permis d'explorer le fonctionnement des slots de réplication et l'impact de la fonction CHECKPOINT sur les fichiers WAL.

Cependant, il est important de souligner que cette solution s'avère difficilement implémentable pour un modèle cloud natif tel que Kubernetes. La gestion complexe du fail-over est un défi notable, bien que des solutions existent, notamment à travers des opérateurs Kubernetes comme celui proposé par Zalando.

En dépit des difficultés rencontrées, ce TP a été enrichissant, offrant des perspectives éclairantes sur les choix à considérer dans des projets futurs. L'expérience acquise souligne l'importance de prendre en compte les spécificités des environnements cloud natifs et des solutions dédiées pour garantir une mise en œuvre efficace de la réplication dans des contextes tels que Kubernetes.