Module R56

Réplication Physique : fondamentaux

24.04

Table des matières

Sur ce document

Formation	Module R56
Titre	Réplication Physique : fondamentaux
Révision	24.04
PDF	https://dali.bo/r56_pdf
EPUB	https://dali.bo/r56_epub
HTML	https://dali.bo/r56_html
Slides	https://dali.bo/r56_slides

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

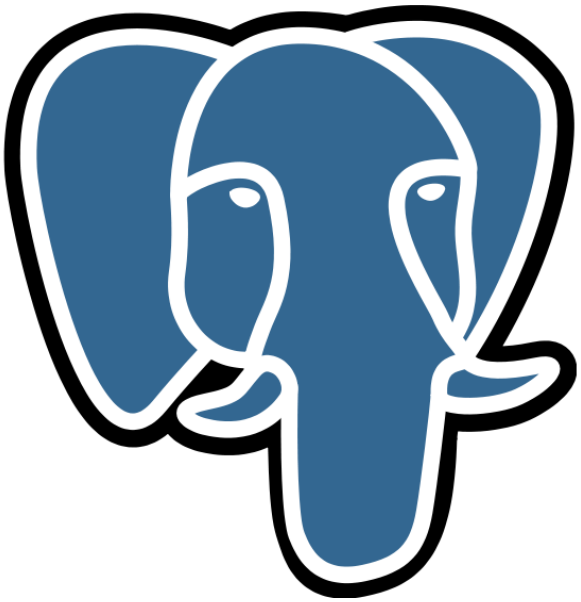
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Réplication physique : rappels



1.1 INTRODUCTION

- Patroni repose sur la réplication physique de PostgreSQL
- la haute disponibilité implique les équipes techniques
- les équipes techniques doivent comprendre la mécanique

Patroni configure la réplication physique native de PostgreSQL pour assurer la redondance des données au sein de l'agrégat. Il est important de bien maîtriser comment cette réplication fonctionne.

Effectivement, Patroni ayant pour but d'optimiser la disponibilité, l'administrateur doit être lui-même réactif et identifier rapidement les causes d'un incident ou d'un problème de réplication, savoir réintégrer un nœud en réplication, etc. Et ce, sans créer de sur-incident bien entendu.

1.1.1 Au menu

- Mise en place de la réplication physique
- Promotion
- Retour à l'état stable

1.2 MISE EN PLACE DE LA RÉPLICATION PAR STREAMING

- Réplication en flux
- Un processus du serveur primaire discute avec un processus du serveur secondaire
  - d'où un *lag* moins important
- Asynchrone ou synchrone
- En cascade

Le serveur PostgreSQL secondaire lance un processus appelé `walreceiver`, dont le but est de se connecter au serveur primaire et d'attendre les modifications de la réplication.

Le `walreceiver` a donc besoin de se connecter sur le serveur PostgreSQL primaire. Ce dernier doit être configuré pour accepter cette connexion. Quand elle est acceptée par le serveur primaire, le serveur PostgreSQL du serveur primaire lance un nouveau processus, appelé `walsender`. Ce dernier a pour but d'envoyer les données de réplication au serveur secondaire. Les données de réplication sont envoyées suivant l'activité et certains paramètres de configuration.

Cette méthode permet une réplication plus proche du serveur primaire que le *log shipping*. On peut même configurer un mode synchrone : un client du serveur primaire ne récupère pas la main tant que ses modifications ne sont pas enregistrées sur le serveur primaire **et** sur le serveur secondaire synchrone. Cela s'effectue à la validation de la transaction, implicite ou lors d'un `COMMIT`.

Enfin, la réplication en cascade permet à un secondaire de fournir les informations de réplication à un autre secondaire, déchargeant ainsi le serveur primaire d'un certain travail et diminuant aussi la bande passante réseau utilisée par le serveur primaire.

1.2.1 Serveur primaire (1/2) - Configuration

Dans `postgresql.conf` :

- `wal_level = replica` (ou `logical`)
- `max_wal_senders = X`
  - 1 par client par *streaming*
  - défaut : 10
- `wal_sender_timeout = 60s`

Il faut tout d'abord s'assurer que PostgreSQL enregistre suffisamment d'informations pour que le serveur secondaire puisse rejouer toutes les modifications survenant sur le serveur primaire. Dans certains cas, PostgreSQL peut économiser l'écriture de journaux quand cela ne pose pas de problème pour l'intégrité des données en cas de crash. Par exemple, sur une instance sans archivage ni réplication, il est inutile de tracer la totalité d'une transaction qui commence par créer une table, puis qui la remplit. En cas de crash pendant l'opération, l'opération complète est annulée, la table n'existera plus : PostgreSQL peut donc écrire directement son contenu sur le disque sans journaliser.

Cependant, pour restaurer cette table ou la répliquer, il est nécessaire d'avoir les étapes intermédiaires (le contenu de la table) et il faut donc écrire ces informations supplémentaires dans les journaux.

Le paramètre `wal_level` fixe le comportement à adopter. Comme son nom l'indique, il permet de préciser le niveau d'informations que l'on souhaite avoir dans les journaux. Il connaît trois valeurs :

- Le niveau `replica` est adapté à l'archivage ou la réplication, en plus de la sécurisation contre les arrêts brutaux. C'est le niveau par défaut. L'optimisation évoquée plus haut n'est pas possible.
- Le niveau `minimal` n'offre que la protection contre les arrêts brutaux, mais ne permet ni réplication ni sauvegarde PITR. Ce niveau ne sert plus guère qu'aux environnements ni archivés, ni répliqués, pour réduire la quantité de journaux générés, comme dans l'optimisation ci-dessus.
- Le niveau `logical` est le plus complet et doit être activé pour l'utilisation du décodage logique, notamment pour utiliser la réplication logique. Il n'est pas nécessaire pour la sauvegarde PITR ou la réplication physique, ni incompatible.

Le serveur primaire accepte un nombre maximum de connexions de réplication : il s'agit du paramètre `max_wal_senders`. Il faut compter au moins une connexion pour chaque serveur secondaire susceptible de se connecter, ou les outils utilisant le *streaming* comme `pg_basebackup` ou `pg_receivewal`. Il est conseillé de prévoir « large » d'entrée : l'impact mémoire est négligeable, et cela évite d'avoir à redémarrer l'instance primaire à chaque modification. La valeur par défaut de 10 devrait suffire dans la plupart des cas.

Le paramètre `wal_sender_timeout` permet de couper toute connexion inactive après le délai indiqué par ce paramètre. Par défaut, le délai est d'une minute. Cela permet au serveur primaire de ne pas conserver une connexion coupée ou dont le client a disparu pour une raison ou une autre. Le secondaire retentera par la suite une connexion complète.

1.2.2 Serveur primaire (2/2) - Authentification

- Le serveur secondaire doit pouvoir se connecter au serveur primaire
- Pseudo-base `replication`
- Utilisateur dédié conseillé avec attributs `LOGIN` et `REPLICATION`
- Configurer `pg_hba.conf` :

```
host replication user_repli 10.2.3.4/32   scram-sha-256
```

- Recharger la configuration

Il est nécessaire après cela de configurer le fichier `pg_hba.conf`. Dans ce fichier, une ligne (par secondaire) doit indiquer les connexions de réplication. L'idée est d'éviter que tout le monde puisse se connecter pour répliquer l'intégralité des données.

Pour distinguer une ligne de connexion standard et une ligne de connexion de réplication, la colonne indiquant la base de données doit contenir le mot « replication ». Par exemple :

```
host    replication    user_repli    10.0.0.2/32    scram-sha-256
```

Dans ce cas, l'utilisateur `user_repli` pourra entamer une connexion de réplication vers le serveur primaire à condition que la demande de connexion provienne de l'adresse IP `10.0.0.2` et que cette demande de connexion précise le bon mot de passe au format `scram-sha-256`.

Un utilisateur dédié à la réplication est conseillé pour des raisons de sécurité. On le créera avec les droits suivants :

```
CREATE ROLE user_repli LOGIN REPLICATION ;
```

et bien sûr un mot de passe complexe.

Les connexions locales de réplication sont autorisées par défaut sans mot de passe.

Après modification du fichier `postgresql.conf` et du fichier `pg_hba.conf`, il est temps de demander à PostgreSQL de recharger sa configuration. L'action `reload` suffit dans tous les cas, sauf celui où `max_wal_senders` est modifié (auquel cas il faudra redémarrer PostgreSQL).

1.2.3 Serveur secondaire (1/4) - Copie des données

Copie des données du serveur primaire (à chaud !) :

- Copie généralement à chaud donc incohérente !
- Le plus simple : `pg_basebackup`
  - simple mais a des limites
- Idéal : outil PITR
- Possible : `rsync`, `cp`…
  - ne pas oublier `pg_backup_start()` / `pg_backup_stop()` !
  - exclure certains répertoires et fichiers
  - garantir la disponibilité des journaux de transaction

La première action à réaliser ressemble beaucoup à ce que propose la sauvegarde en ligne des fichiers. Il s'agit de copier le répertoire des données de PostgreSQL ainsi que les tablespaces associés.

Rappelons que généralement cette copie aura lieu à chaud, donc une simple copie directe sera incohérente.

pg_basebackup :

L'outil le plus simple est `pg_basebackup`. Ses avantages sont sa disponibilité et sa facilité d'utilisation. Il sait ce qu'il n'y a pas besoin de copier et peut inclure les journaux nécessaires pour ne pas avoir à paramétrer l'archivage.

Il peut utiliser la connexion de réplication déjà prévue pour le secondaire, poser des slots temporaires ou le slot définitif.

Pour faciliter la mise en place d'un secondaire, il peut générer les fichiers de configuration à partir des paramètres qui lui ont été fournis (option `--write-recovery-conf`).

Malgré beaucoup d'améliorations dans les dernières versions, la limite principale de `pg_basebackup` reste d'exiger un répertoire cible vide : on doit toujours recopier l'intégralité de la base copiée. Cela peut être pénible lors de tests répétés avec une grosse base, ou avec une liaison instable.

Outils PITR :

L'idéal est un outil de restauration PITR permettant la restauration en mode delta, par exemple pgBackRest avec l'option `--delta`. Ne sont restaurés que les fichiers ayant changé, et le primaire n'est pas chargé par la copie.

rsync :

Un script de copie reste une option possible. Il est possible de le faire manuellement, tout comme pour une sauvegarde PITR.

Une copie manuelle implique que les journaux sont archivés par ailleurs.

Rappelons les trois étapes essentielles :

- le `pg_backup_start()` ;
- la copie des fichiers : généralement avec `rsync --whole-file`, ou tout moyen permettant une copie fiable et rapide ;
- le `pg_backup_stop()`.

On exclura les fichiers inutiles lors de la copie qui pourraient gêner un redémarrage, notamment le fichier `postmaster.pid` et les répertoires `pg_wal`, `pg_replslot`, `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp`, `pg_subtrans`, `pgslq_tmp*`. La liste complète figure dans la documentation officielle[1].

1.2.4 Serveur secondaire (2/4) - Fichiers de configuration

- `postgresql.conf` & `postgresql.auto.conf`
  - paramètres
- `standby.signal` (dans PGDATA)
  - vide

Au choix, les paramètres sont à ajouter dans `postgresql.conf`, dans un fichier appelé par ce dernier avec une clause d'inclusion, ou dans `postgresql.auto.conf` (forcément dans le répertoire de données pour ce dernier, et qui surcharge les fichiers précédents). Cela dépend des habitudes, de la méthode d'industrialisation…

S'il y a des paramètres propres au primaire dans la configuration d'un secondaire, ils seront ignorés, et vice-versa. Dans les cas simples, le `postgresql.conf` peut donc être le même.

Puis il faut créer un fichier vide nommé `standby.signal` dans le répertoire `PGDATA`, qui indique à PostgreSQL que le serveur doit rester en *recovery* permanent.

[1] https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP

Au cas où vous rencontreriez un vieux serveur en version antérieure à la 12 : jusqu'en version 11, on activait le mode *standby* non dans la configuration, mais en créant un fichier texte `recovery.conf` dans le `PGDATA` de l'instance, et en y plaçant le paramètre `standby_mode` à `on`. Les autres paramètres sont les mêmes. Toute modification impliquait un redémarrage.

1.2.5 Serveur secondaire (2/4) - Paramètres

- `primary_conninfo` (*streaming*) :

```
primary_conninfo = 'user=postgres host=prod port=5434
 passfile=/var/lib/postgresql/.pgpass
 application_name=secondaire2 '
```

- Optionnel :
  - `primary_slot_name`
  - `recovery_command`
  - `wal_receiver_timeout`

PostgreSQL doit aussi savoir comment se connecter au serveur primaire. C'est le paramètre `primary_conninfo` qui le lui dit. Il s'agit d'un DSN standard où il est possible de spécifier l'adresse IP de l'hôte ou son alias, le numéro de port, le nom de l'utilisateur, etc. Il est aussi possible de spécifier le mot de passe, mais c'est risqué en terme de sécurité. En effet, PostgreSQL ne vérifie pas si ce fichier est lisible par quelqu'un d'autre que lui. Il est donc préférable de placer le mot de passe dans le fichier `.pgpass`, généralement dans `~postgres/` sur le secondaire, fichier qui n'est utilisé que s'il n'est lisible que par son propriétaire. Par exemple :

```
primary_conninfo = 'user=postgres host=prod passfile=/var/lib/postgresql/.pgpass'
```

Toutes les options de la libpq sont accessibles. Par exemple, cette chaîne de connexion a été générée pour un nouveau secondaire par `pg_basebackup -R` :

```
primary_conninfo = 'host=prod user=postgres passfile=''/var/lib/postgresql/.pgpass''
 ↪  channel_binding=prefer port=5436 sslmode=prefer sslcompression=0
 ↪  sslcertmode=allow sslsni=1 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer
 ↪  krbsrvname=postgres gssdelegation=0 target_session_attrs=any
 ↪  load_balance_hosts=disable
```

S'y trouvent beaucoup de paramétrage par défaut dépendant de méthodes d'authentification, ou pour le SSL.

Parmi les autres paramètres optionnels de `primary_conninfo`, il est conseillé d'ajouter `application_name`, par exemple avec le nom du serveur. Cela facilite la supervision. C'est même nécessaire pour paramétrer une réplication synchrone.

```
primary_conninfo = 'user=postgres host=prod passfile=/var/lib/postgresql/.pgpass
↪  application_name=secondaire2 '
```

Si `application_name` n'est pas fourni, le `cluster_name` du secondaire sera utilisé, mais il est rarement correctement configuré (par défaut, il vaut `16/main` sur Debian/Ubuntu, et n'est pas configuré sur Red Hat/Rocky Linux).

De manière optionnelle, nous verrons que l'on peut définir aussi deux paramètres :

- `primary_slot_name`, pour sécuriser la réplication avec un slot de réplication ;
- `restore_command`, pour sécuriser la réplication avec un accès à la sauvegarde PITR.

Le paramètre `wal_receiver_timeout` sur le secondaire est le symétrique de `wal_sender_timeout` sur le primaire. Il indique au bout de combien de temps couper une connexion inactive. Le secondaire retentera la connexion plus tard.

1.2.6 Serveur secondaire (3/4) - Démarrage

- Démarrer PostgreSQL
- Suivre dans les traces que tout va bien

Il ne reste plus qu'à démarrer le serveur secondaire.

En cas de problème, le premier endroit où aller chercher est bien entendu le fichier de trace `postgresql.log`.

1.2.7 Processus

Sur le primaire :

- `walsender ... streaming 0/3BD48728`

Sur le secondaire :

- `walreceiver   streaming 0/3BD48728`

Sur le primaire, un processus `walsender` apparaît pour chaque secondaire connecté. Son nom de processus est mis à jour en permanence avec l'emplacement dans le flux de journaux de transactions :

```
postgres: 16/secondaire1: walsender postgres [local] streaming 15/6A6EF408
postgres: 16/secondaire2: walsender postgres [local] streaming 15/6A6EF408
```

Symétriquement, sur chaque secondaire, un process `walreceiver` apparaît.

```
postgres: 16/secondaire2: walreceiver streaming 0/DD73C218
```

1.3 PROMOTION

- Attention au *split-brain* !
- Vérification avant promotion
- Promotion : méthode et déroulement
- Retour à l'état stable

1.3.1 Attention au split-brain !

- Si un serveur secondaire devient le nouveau primaire
  - s'assurer que l'ancien primaire ne reçoit plus d'écriture
- Éviter que les deux instances soient ouvertes aux écritures
  - confusion et perte de données !

La pire chose qui puisse arriver lors d'une bascule est d'avoir les deux serveurs, ancien primaire et nouveau primaire promu, ouverts tous les deux en écriture. Les applications risquent alors d'écrire dans l'un ou l'autre…

Quelques histoires « d'horreur » à ce sujet :

- de nombreux exemples sur diverses technologies de réplication[2] ;
- *post mortem* d'un gros problème chez Github en 2018[3].

[2] https://github.blog/2018-10-30-oct21-post-incident-analysis
[3] https://aphyr.com/posts/288-the-network-is-reliable

1.3.2 Vérification avant promotion

- Primaire :

```
# systemctl stop postgresql-14

$ pg_controldata -D /var/lib/pgsql/14/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state:               shut down
Latest checkpoint's REDO location:    0/3BD487D0
```

- Secondaire :

```
$ psql -c 'CHECKPOINT;'
$ pg_controldata -D /var/lib/pgsql/14/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/3BD487D0
```

Avant une bascule, il est capital de vérifier que toutes les modifications envoyées par le primaire sont arrivées sur le secondaire. Si le primaire a été arrêté proprement, ce sera le cas. Après un `CHECKPOINT` sur le secondaire, on y retrouvera le même emplacement dans les journaux de transaction.

Ce contrôle doit être systématique avant une bascule. Même si toutes les écritures applicatives sont stoppées sur le primaire, quelques opérations de maintenance peuvent en effet écrire dans les journaux et provoquer un écart entre les deux serveurs (divergence). Il n'y aura alors pas de perte de données mais cela pourrait gêner la transformation de l'ancien primaire en secondaire, par exemple.

Noter que `pg_controldata` n'est pas dans les chemins par défaut des distributions. La fonction SQL `pg_control_checkpoint()` affiche les même informations, mais n'est bien sûr pas accessible sur un primaire arrêté.

1.3.3 Promotion du standby : méthode

- Shell :
  - `pg_ctl promote`
- SQL :
  - fonction `pg_promote()`
- Déclenchement par fichier :
  - `promote_trigger_file` (<=v15)

Il existe plusieurs méthodes pour promouvoir un serveur PostgreSQL en mode *standby*. Les méthodes les plus appropriées sont :

- l'action `promote` de l'outil `pg_ctl`, ou de son équivalent dans les scripts des paquets d'installation, comme `pg_ctlcluster` sous Debian ;
- la fonction SQL `pg_promote`.

Ces deux méthodes remplacent le fichier de déclenchement historique (*trigger file*), défini par le paramètre `promote_trigger_file`, qui n'existe plus à partir de PostgreSQL 16. Dans les versions précédentes, un serveur secondaire vérifie en permanence si ce fichier existe. Dès qu'il apparaît, l'instance est promue. Par mesure de sécurité, il est préconisé d'utiliser un emplacement accessible uniquement aux administrateurs.

1.3.4 Promotion du standby : déroulement

Une promotion déclenche :

- déconnexion de la *streaming replication* (bascule programmée)
- rejeu des dernières transactions en attente d'application
- choix d'une nouvelle *timeline* du journal de transaction
- suppression du fichier `standby.signal`
- ouverture aux écritures

Une fois le serveur promu, il finit de rejouer les données de transaction en provenance du serveur principal en sa possession et se déconnecte de celui-ci s'il est configuré en *streaming replication*.

Ensuite, il choisit une nouvelle *timeline* pour son journal de transactions. La timeline est le premier numéro dans le nom du segment (fichier WAL), soit par exemple une timeline 5 pour un fichier nommé `000000050000003200000031`).

Enfin, il autorise les connexions en lecture et en écriture.

Comme le serveur reçoit à présent des modifications différentes du serveur principal qu'il répliquait précédemment, il ne peut être reconnecté à ce serveur. Le choix d'une nouvelle *timeline* permet à PostgreSQL de rendre les journaux de transactions de ce nouveau serveur en écriture incompatibles avec son ancien serveur principal. De plus, créer des journaux de transactions avec un nom de fichier différent rend possible l'archivage depuis ce nouveau serveur en écriture sans perturber l'ancien. Il n'y a pas de fichiers en commun même si l'espace d'archivage est partagé.

Les *timelines* ne changent pas que lors des promotions, mais aussi lors des restaurations PITR. En général, on désire que les secondaires (parfois en cascade) suivent. Heureusement, ceci est le paramétrage par défaut depuis la version 12 :

```
recovery_target_timeline = latest
```

1.3.5 Opérations après promotion du standby

- `VACUUM ANALYZE` conseillé
  - calcul d'informations nécessaires pour autovacuum

Il n'y a aucune opération obligatoire après une promotion. Cependant, il peut être intéressant d'exécuter un `VACUUM` ou un `ANALYZE` pour que PostgreSQL mette à jour les estimations de nombre de lignes vivantes et mortes. Ces estimations sont utilisées par l'autovacuum pour lutter contre la fragmentation des tables et mettre à jour les statistiques sur les données. Or ces estimations faisant partie des statistiques d'activité, elles ne sont pas répliquées vers les secondaires. Il est donc intéressant de les mettre à jour après une promotion.

1.3.6 Retour à l'état stable

- Si un *standby* a été momentanément indisponible, reconnexion directe possible si :
  - journaux nécessaires encore présents sur primaire (slot, `wal_keep_size` / `wal_keep_segments`)
  - journaux nécessaires présents en archives (`restore_command`)
- Sinon
  - « décrochage »
  - reconstruction nécessaire

Si un serveur secondaire est momentanément indisponible mais revient en ligne sans perte de données (réseau coupé, problème OS…), alors il a de bonnes chances de se « raccrocher » à son serveur primaire. Il faut bien sûr que l'ensemble des journaux de transaction depuis son arrêt soit accessible à ce serveur, sans exception.

En cas de réplication par *streaming* : le primaire ne doit pas avoir recyclé les journaux après ses *checkpoints*. Il les aura conservés s'il y a un slot de réplication actif dédié à ce secondaire, ou si on a monté `wal_keep_size` (ou `wal_keep_segments` jusque PostgreSQL 12 compris) assez haut par rapport à l'activité en écriture sur le primaire. Les journaux seront alors toujours disponibles sur le principal et le secondaire rattrapera son retard par *streaming*. Si le primaire n'a plus les journaux, il affichera une erreur, et le secondaire tentera de se rabattre sur le *log shipping*, s'il est aussi configuré.

En cas de réplication par *log shipping*, il faut que la `restore_command` fonctionne, que le stock des journaux remonte assez loin dans le temps (jusqu'au moment où le secondaire a perdu contact), et qu'aucun journal ne manque ou ne soit corrompu. Sinon le secondaire se bloquera au dernier journal chargé. En cas d'échec, ou si le dernier journal disponible vient d'être rejoué, le secondaire basculera sur le *streaming*, s'il est configuré.

Si le secondaire ne peut rattraper le flux des journaux du primaire, il doit être reconstruit par l'une des méthodes précédentes.

1.3.7 Retour à l'état stable, suite

- Synchronisation automatique une fois la connexion rétablie
- Mais reconstruction obligatoire :
  - si le serveur secondaire était plus avancé que le serveur promu (« divergence »)
- Reconstruire les serveurs secondaires à partir du nouveau principal :
  - `rsync`, restauration PITR, plutôt que `pg_basebackup`
  - `pg_rewind`
- Reconstruction : manuelle !
- Tablespaces !

Un secondaire qui a bien « accroché » son primaire se synchronise automatiquement avec lui, que ce soit par *streaming* ou *log shipping*. C'est notamment le cas si l'on vient de le construire depuis une sauvegarde ou avec `pg_basebackup`, et que l'archivage ou le *streaming* alimentent correctement le secondaire. Cependant, il y a des cas où un secondaire ne peut être simplement raccroché à un primaire, notamment si le secondaire se croit plus avancé que le primaire dans le flux des journaux.

Le cas typique est un ancien primaire que l'on veut transformer en secondaire d'un ancien secondaire promu. Si la bascule s'était faite proprement, et que l'ancien primaire avait pu envoyer tous ses journaux avant de s'arrêter ou d'être arrêté, il n'y a pas de problème. Si le primaire a été arrêté violemment, sans pouvoir transmettre tous ses journaux, l'ancien secondaire n'a rejoué que ce qu'il a reçu, puis a ouvert en écriture sa propre *timeline* depuis un point moins avancé que là où le primaire était finalement arrivé avant d'être arrêté. Les deux serveurs ont donc « divergé », même pendant très peu de temps. Les journaux non envoyés au nouveau primaire doivent être considérés comme perdus. Quand l'ancien primaire revient en ligne, parfois très longtemps après, il voit que sa *timeline* est plus avancée que la version qu'en a gardée le nouveau primaire. Il ne sait donc pas comment appliquer les journaux qu'il reçoit du nouveau primaire.

La principale solution, et la plus simple, reste alors la reconstruction du secondaire à raccrocher.

L'utilisation de `pg_basebackup` est possible mais déconseillée si la volumétrie est importante : cet outil impose une copie de l'ensemble des données du serveur principal, et ce peut être long.

La durée de reconstruction des secondaires peut être optimisée en utilisant des outils de synchronisation de fichiers pour réduire le volume des données à transférer. Les outils de restauration PITR offrent souvent une restauration en mode delta (notamment l'option `--delta` de pgBackRest) et c'est ce qui est généralement à privilégier. Dans un script de sauvegarde PITR, `rsync --whole-file` reste une bonne option.

Le fait de disposer de l'ensemble des fichiers de configuration sur tous les nœuds permet de gagner

un temps précieux lors des phases de reconstruction, qui peuvent également être scriptées.

Par contre, les opérations de reconstructions se doivent d'être lancées **manuellement** pour éviter tout risque de corruption de données dues à des opérations automatiques externes, comme lors de l'utilisation de solutions de haute disponibilité.

Enfin, on rappelle qu'il ne faut pas oublier de prendre en compte les *tablespaces* lors de la reconstruction.

Une alternative à la reconstruction est l'utilisation de l'outil `pg_rewind` pour « rembobiner » l'ancien primaire, si tous les journaux nécessaires sont disponibles.

1.4 CONCLUSION

- La réplication physique native est très robuste
- Patroni peut automatiser :
  - la mise en réplication
  - la promotion
  - le raccrochage d'un ancien primaire

La robustesse de la réplication physique est éprouvée depuis longtemps. C'est à cette base solide que Patroni amène l'automatisation de :

- l'ajout de nœuds supplémentaires ;
- la promotion automatique en fonction du nœud ;
- le raccrochage d'un ancien primaire au nouveau primaire.

Ces mécanismes sont abordé dans le module consacré à Patroni[4].

[4] https://dali.bo/r58_html

1.5 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

1.5.1 Sur Rocky Linux 8 ou 9

ATTENTION : Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est fortement conseillée.

Installation du dépôt communautaire :

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent sont inspirées de celles générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 16) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 16 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql16-server postgresql16-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql16-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb
# cat /var/lib/pgsql/16/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-16/bin/pg_ctl -D /var/lib/pgsql/16/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-16/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/16/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier lancement.

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-16
# systemctl stop postgresql-16
# systemctl status postgresql-16
# systemctl reload postgresql-16
# systemctl restart postgresql-16
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au lancement du système d'exploitation :

```
# systemctl enable postgresql-16
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

Si plusieurs instances d'une *même version majeure* (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faut les installer dans des `PGDATA` différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/16/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-16.service \
        /etc/systemd/system/postgresql-16-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/16/infocentre
```

- Option 1 : création d'une nouvelle instance vierge :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb postgresql-16-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/16/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-16-infocentre
# systemctl [enable|disable] postgresql-16-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

1.5.2 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Référence : https://apt.postgresql.org/

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 16 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-16 postgresql-client-16
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/16/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/16/main`
Fichiers de configuration	dans `/etc/postgresql/16/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 16 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/16/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 16 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- créer une instance :

```
# pg_createcluster 16 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 16 infocentre \
  --port=12345 \
  --datadir=/PGDATA/16/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/16/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/16/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 16 infocentre start
```

1.5.3 Accès à l'instance depuis le serveur même (toutes distributions)

Par défaut, l'instance n'est accessible que par l'utilisateur système **postgres**, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (16.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (16.0)
Type "help" for help.
postgres=#
```

- Pour se connecter sans taper le mot de passe à une instance, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

Ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- Pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/16/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-16

root:~# pg_ctlcluster 16 main reload

postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

1.6 TRAVAUX PRATIQUES

Ce TP suppose que les instances tournent sur la même machine. N'oubliez pas qu'il faut un répertoire de données et un numéro de port par serveur PostgreSQL.
Dans la réalité, il s'agira de deux machines différentes : l'archivage nécessitera des opérations supplémentaires (montage de partitions réseau, connexion ssh sans mot de passe…).

1.6.1 Sur Rocky Linux 8 ou 9

1.6.1.1 Réplication asynchrone en flux avec un seul secondaire

But : Mettre en place une réplication asynchrone en flux.

- Créer l'instance principale dans `/var/lib/pgsql/16/instance1`.

- Mettre en place la configuration de la réplication par *streaming*.
- L'utilisateur dédié sera nommé **repli**.

- Créer la première instance secondaire **instance2**, par copie **à chaud** du répertoire de données avec `pg_basebackup` vers `/var/lib/psql/16/instance2`.
- Penser à modifier le port de cette nouvelle instance avant de la démarrer.

- Démarrer **instance2** et s'assurer que la réplication fonctionne bien avec `ps`.
- Tenter de se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

1.6.1.2 Promotion de l'instance secondaire

But : Promouvoir un serveur secondaire en primaire.

- En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

- Tenter de se connecter au serveur secondaire fraîchement promu.
- Les écritures y sont-elles possibles ?

1.6.1.3 Retour à la normale

But : Revenir à l'architecture d'origine.

- Reconstruire l'instance initiale (`/var/lib/pgsql/16/instance1`) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2** en utilisant `pg_basebackup`.

- Démarrer cette nouvelle instance.
- Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

- Inverser à nouveau les rôles des deux instances afin que **instance2** redevienne l'instance secondaire.

1.6.2 Sur Debian 12

Ce TP suppose que les instances tournent sur la même machine. N'oubliez pas qu'il faut un répertoire de données et un numéro de port par serveur PostgreSQL.
Dans la réalité, il s'agira de deux machines différentes : l'archivage nécessitera des opérations supplémentaires (montage de partitions réseau, connexion ssh sans mot de passe…).

1.6.2.1 Réplication asynchrone en flux avec un seul secondaire

But : Mettre en place une réplication asynchrone en flux.

- Créer l'instance principale en utilisant `pg_createcluster`.

- Vérifier la configuration de la réplication par *streaming*.
- L'utilisateur dédié sera nommé **repli**.

- Créer la première instance secondaire **instance2**, par copie **à chaud** du répertoire de données avec `pg_basebackup` vers `/var/lib/postgresql/16/instance2`. Le répertoire dédié aux fichiers de configuration devra être copié.
- Penser à modifier les chemins et le n° de port de cette nouvelle instance avant de la démarrer.

- Démarrer **instance2** et s'assurer que la réplication fonctionne bien avec `ps`.
- Tenter de se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

1.6.2.2 Promotion de l'instance secondaire

But : Promouvoir un serveur secondaire en primaire.

- En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

- Tenter de se connecter au serveur secondaire fraîchement promu.
- Les écritures y sont-elles possibles ?

1.6.2.3 Retour à la normale

But : Revenir à l'architecture d'origine.

- Reconstruire l'instance initiale (`/var/lib/postgresql/16/instance1`) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2** en utilisant `pg_basebackup`.

- Démarrer cette nouvelle instance.
- Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

- Inverser à nouveau les rôles des deux instances afin que **instance2** redevienne l'instance secondaire.

1.7 TRAVAUX PRATIQUES (SOLUTIONS)

La version de PostgreSQL est la version 16. Adapter au besoin pour une version ultérieure. Noter que les versions 12 et précédentes utilisent d'autres fichiers.

1.7.1 Sur Rocky Linux 8 ou 9

Cette solution se base sur un système Rocky Linux 8, installé à minima depuis les paquets du PGDG, et en anglais.

Le prompt `#` indique une commande à exécuter avec l'utilisateur `root`. Le prompt `$` est utilisé pour les commandes de l'utilisateur `postgres`.

La mise en place d'une ou plusieurs instances sur le même poste est décrite plus haut.

En préalable, nettoyer les instances précédemment créées sur le serveur.

Ensuite, afin de réaliser l'ensemble des TP, configurer 4 services PostgreSQL « instance[1-4] ».

```
# cp /lib/systemd/system/postgresql-16.service \
                          /etc/systemd/system/instance1.service

# sed -i "s|/var/lib/pgsql/16/data/|/var/lib/pgsql/16/instance1/|" \
                          /etc/systemd/system/instance1.service

# cp /lib/systemd/system/postgresql-16.service \
                          /etc/systemd/system/instance2.service

# sed -i "s|/var/lib/pgsql/16/data/|/var/lib/pgsql/16/instance2/|" \
                          /etc/systemd/system/instance2.service

# cp /lib/systemd/system/postgresql-16.service \
                          /etc/systemd/system/instance3.service

# sed -i "s|/var/lib/pgsql/16/data/|/var/lib/pgsql/16/instance3/|" \
                          /etc/systemd/system/instance3.service

# cp /lib/systemd/system/postgresql-16.service \
                          /etc/systemd/system/instance4.service

# sed -i "s|/var/lib/pgsql/16/data/|/var/lib/pgsql/16/instance4/|" \
                          /etc/systemd/system/instance4.service

-->
```

1.7.1.1 Réplication asynchrone en flux avec un seul secondaire

- Créer l'instance principale dans `/var/lib/pgsql/16/instance1`.

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums'
# /usr/pgsql-16/bin/postgresql-16-setup initdb instance1
Initializing database ... OK
# systemctl start instance1
```

- Mettre en place la configuration de la réplication par *streaming*.
- L'utilisateur dédié sera nommé **repli**.

Depuis la version 10, le comportement de PostgreSQL a changé et la réplication est activée par défaut en local.

Nous allons cependant modifier le fichier `/var/lib/pgsql/16/instance1/pg_hba.conf` pour que l'accès en réplication soit autorisé pour l'utilisateur **repli** :

```
host  replication  repli  127.0.0.1/32  md5
```

Cette configuration indique que l'utilisateur **repli** peut se connecter en mode réplication à partir de l'adresse IP `127.0.0.1`. L'utilisateur `repli` n'existant pas, il faut le créer (nous utiliserons le mot de passe **confidentiel**) :

```
$ createuser --no-superuser --no-createrole --no-createdb --replication -P repli
Enter password for new role:
Enter it again:
```

Configurer ensuite le fichier `.pgpass` de l'utilisateur système `postgres` :

```
$ echo "*:*:*:repli:confidentiel" > ~/.pgpass
$ chmod 600 ~/.pgpass
```

Pour prendre en compte la configuration, la configuration de l'instance principale doit être rechargée :

```
$ psql -c 'SELECT pg_reload_conf()'
```

- Créer la première instance secondaire **instance2**, par copie **à chaud** du répertoire de données avec `pg_basebackup` vers `/var/lib/psql/16/instance2`.
- Penser à modifier le port de cette nouvelle instance avant de la démarrer.

Utiliser `pg_basebackup` pour créer l'instance secondaire :

```
$ pg_basebackup -D /var/lib/pgsql/16/instance2 -P -R -c fast -h 127.0.0.1 -U repli

25314/25314 kB (100%), 1/1 tablespace
```

L'option `-R` ou `--write-recovery-conf` de `pg_basebackup` a préparé la configuration de la mise en réplication en créant le fichier `standby.signal` ainsi qu'en configurant `primary_conninfo` dans le fichier `postgresql.auto.conf` (dans les versions antérieures à la 11, il renseignerait `recovery.conf`) :

```
$ cat /var/lib/pgsql/16/instance2/postgresql.auto.conf

primary_conninfo = 'user=repli passfile=''/var/lib/pgsql/.pgpass''
                    host=127.0.0.1 port=5432 sslmode=prefer sslcompression=0
                    gssencmode=prefer krbsrvname=postgres target_session_attrs=any'

$ ls /var/lib/pgsql/16/instance2/standby.signal

/var/lib/pgsql/16/instance2/standby.signal
```

Il faut désormais positionner le port d'écoute dans le fichier de configuration, c'est-à-dire `/var/lib/pgsql/16/instance2/postgresql.conf` :

```
port=5433
```

- Démarrer **instance2** et s'assurer que la réplication fonctionne bien avec `ps`.
- Tenter de se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

Il ne reste désormais plus qu'à démarrer l'instance secondaire :

```
# systemctl start instance2
```

La commande `ps` suivante permet de voir que les deux serveurs sont lancés :

```
$ ps -o pid,cmd fx
```

La première partie concerne le serveur secondaire :

```
PID CMD
9671 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance2/
9673  \_ postgres: logger
9674  \_ postgres: startup   recovering 000000010000000000000003
9675  \_ postgres: checkpointer
9676  \_ postgres: background writer
9677  \_ postgres: stats collector
9678  \_ postgres: walreceiver   streaming 0/3000148
```

La deuxième partie concerne le serveur principal :

```
PID CMD
9564 /usr/pgsql-16/bin/postmaster -D /var/lib/pgsql/16/instance1/
9566  \_ postgres: logger
9568  \_ postgres: checkpointer
9569  \_ postgres: background writer
9570  \_ postgres: walwriter
9571  \_ postgres: autovacuum launcher
9572  \_ postgres: stats collector
9573  \_ postgres: logical replication launcher
9679  \_ postgres: walsender repli 127.0.0.1(58420) streaming 0/3000148
```

Pour différencier les deux instances, il est possible d'identifier le répertoire de données (l'option `-D`), les autres processus sont des fils du processus postmaster. Il est aussi possible de configurer le paramètre `cluster_name`.

Nous avons bien les deux processus de réplication en flux `wal sender` et `wal receiver`.

Créons quelques données sur le principal et assurons-nous qu'elles soient transmises au secondaire :

```
$ createdb b1

$ psql b1

psql (16.1)
Type "help" for help.

b1=# CREATE TABLE t1(id integer);

CREATE TABLE

b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);

INSERT 0 1000000
```

On constate que le flux a été transmis :

```
b1=# \! ps -o pid,cmd fx | egrep "(startup|walsender|walreceiver)"

 9674  \_ postgres: startup   recovering 000000010000000000000006
 9678  \_ postgres: walreceiver   streaming 0/6D4CD28
 9679  \_ postgres: walsender repli 127.0.0.1(58420) streaming 0/6D4CD28
[...]
```

Essayons de nous connecter au secondaire et d'exécuter quelques requêtes :

```
$ psql -p 5433 b1

psql (16.1)
Type "help" for help.

b1=# SELECT COUNT(*) FROM t1;

  count
---------
 1000000

b1=# CREATE TABLE t2(id integer);

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

On peut se connecter, lire des données, mais pas écrire.

Le comportement est visible dans les logs de l'instance secondaire dans le répertoire `/var/lib/pgsql/16/instance2/log` :

```
... LOG:  database system is ready to accept read only connections
```

PostgreSQL indique bien qu'il accepte des connexions en lecture seule.

1.7.1.2 Promotion de l'instance secondaire

- En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

Arrêt de l'instance primaire et vérification de son état :

```
# systemctl stop instance1
```

```
$ /usr/pgsql-16/bin/pg_controldata -D /var/lib/pgsql/16/instance1/ \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               shut down
Latest checkpoint's REDO location:    0/6D4E5C8
```

Vérification de l'instance secondaire :

```
$ psql -p 5433 -c 'CHECKPOINT;'
```

```
$ /usr/pgsql-16/bin/pg_controldata -D /var/lib/pgsql/16/instance2/ \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/6D4E5C8
```

L'instance principale est bien arrêtée, l'instance secondaire est bien en `archive recovery` et les deux sont bien synchronisées.

Promotion de l'instance secondaire :

```
$ /usr/pgsql-16/bin/pg_ctl -D /var/lib/pgsql/16/instance2 promote
```

```
waiting for server to promote.... done
server promoted
```

- Tenter de se connecter au serveur secondaire fraîchement promu.
- Les écritures y sont-elles possibles ?

Connectons-nous à ce nouveau primaire et tentons d'y insérer des données :

```
$ psql -p 5433 b1
psql (16.1)
Type "help" for help.
```

```
b1=# CREATE TABLE t2(id integer);
```

```
CREATE TABLE
```

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

Les écritures sont désormais bien possible sur cette instance.

1.7.1.3 Retour à la normale

- Reconstruire l'instance initiale (`/var/lib/pgsql/16/instance1`) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2** en utilisant `pg_basebackup`.

Afin de rétablir la situation, nous pouvons réintégrer l'ancienne instance primaire en tant que nouveau secondaire. Pour ce faire, nous devons re-synchroniser les données. Utilisons `pg_basebackup` comme précédemment après avoir mis de côté les fichiers de l'ancien primaire :

```
$ mv /var/lib/pgsql/16/instance1 /var/lib/pgsql/16/instance1.old
```

```
$ pg_basebackup -D /var/lib/pgsql/16/instance1 -P -R -c fast \
-h 127.0.0.1 -p 5433 -U repli
```

```
104385/104385 kB (100%), 1/1 tablespace
```

Créer le fichier `standby.signal` s'il n'existe pas déjà. Contrôler `postgresql.auto.conf` (qui contient potentiellement deux lignes `primary_conninfo` !) et adapter le port :

```
$ touch /var/lib/pgsql/16/instance1/standby.signal
```

```
$ cat /var/lib/pgsql/16/instance1/postgresql.auto.conf
```

```
primary_conninfo = 'user=repli passfile=''/var/lib/pgsql/.pgpass'' host=127.0.0.1
 ↪  port=5433 sslmode=prefer sslcompression=0 gssencmode=prefer krbsrvname=postgres
 ↪  target_session_attrs=any'
```

Repositionner le port d'écoute dans le fichier `/var/lib/pgsql/16/instance1/postgresql.conf` :

```
port=5432
```

Enfin, démarrer le service :

```
# systemctl start instance1
```

- Démarrer cette nouvelle instance.
- Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

Les processus adéquats sont bien présents :

```
$ ps -o pid,cmd fx | egrep "(startup|walsender|walreceiver)"
```

```
12520  \_ postgres: startup   recovering 00000002000000000000000A
12524  \_ postgres: walreceiver   streaming 0/A000148
12525  \_ postgres: walsender repli 127.0.0.1(38614) streaming 0/A000148
```

```
$ psql -p 5432 b1
```

```
psql (16.1)
Type "help" for help.
```

En nous connectant à la nouvelle instance secondaire (port 5432), vérifions que les données précédemment insérées dans la table `t2` sont bien présentes :

```
b1=# SELECT COUNT(*) FROM t2;

  count
---------
 1000000
```

> – Inverser à nouveau les rôles des deux instances afin que **instance2** redevienne l'instance secondaire.

Afin que l'instance 5432 redevienne primaire et celle sur le port 5433 secondaire, on peut ré-appliquer la procédure de promotion vue précédemment dans l'autre sens.

Arrêt de l'instance primaire et vérification de son état :

```
# systemctl stop instance2

$ /usr/pgsql-16/bin/pg_controldata -D /var/lib/pgsql/16/instance2/ \
| grep -E '(cluster)|(REDO)'

Database cluster state:               shut down
Latest checkpoint's REDO location:    0/C000060
```

Vérification de l'instance secondaire :

```
$ psql -p 5432 -c 'CHECKPOINT;'
$ /usr/pgsql-16/bin/pg_controldata -D /var/lib/pgsql/16/instance1/ \
| grep -E '(cluster)|(REDO)'

Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/C000060
```

L'instance principale est bien arrêtée, l'instance secondaire est bien en `archive recovery` et les deux sont bien synchronisées.

Promotion de l'instance secondaire :

```
$ /usr/pgsql-16/bin/pg_ctl -D /var/lib/pgsql/16/instance1 promote

waiting for server to promote.... done
server promoted
```

Afin que **instance2** redevienne l'instance secondaire, créer le fichier `standby.signal`, démarrer le service et vérifier que les processus adéquats sont bien présents :

```
$ touch /var/lib/pgsql/16/instance2/standby.signal

# systemctl start instance2

$  ps -o pid,cmd fx | egrep "(startup|walsender|walreceiver)"

 5844  \_ postgres: startup   recovering 00000003000000000000000C
 5848  \_ postgres: walreceiver   streaming 0/C0001F0
 5849  \_ postgres: walsender repli 127.0.0.1(48230) streaming 0/C0001F0
```

1.7.2 Sur Debian 12

Cette solution se base sur un système Debian 12, installé à minima depuis les paquets du PGDG, et en anglais.

Le prompt `#` indique une commande à exécuter avec l'utilisateur `root`. Le prompt `$` est utilisé pour les commandes de l'utilisateur `postgres`.

La mise en place d'une ou plusieurs instances sur le même poste est décrite plus haut.

En préalable, nettoyer les instances précédemment créées sur le serveur.

```
# pg_dropcluster --stop 16 main
# pg_dropcluster --stop 16 infocentre
```

1.7.2.1 Réplication asynchrone en flux avec un seul secondaire

- Créer l'instance1, qui sera l'instance principal.

```
# pg_createcluster 16 instance1
Creating new PostgreSQL cluster 16/instance1 ...
...
Ver Cluster   Port Status Owner    Data directory                  //
16  instance1 5432 down   postgres /var/lib/postgresql/16/instance1 //

Log file
/var/log/postgresql/postgresql-16-instance1.log
```

Le répertoire des données se trouvera sous `/var/lib/postgresql/16/instance1`.

Démarrer l'instance, soit avec :

```
# pg_ctlcluster start 16 instance1
```

soit explicitement via systemd :

```
# systemctl start postgresql@16-instance1
```

- Vérifier la configuration de la réplication par *streaming*.
- L'utilisateur dédié sera nommé **repli**.

Depuis la version 10, le comportement de PostgreSQL a changé et la réplication est activée par défaut en local.

Au sein du fichier `/etc/postgresql/16/instance1/pg_hba.conf`, l'entrée ci-dessous montre que tout utilisateur authentifié (avec l'attribut *REPLICATION*) aura accès en réplication à l'instance :

```
host    replication     all             127.0.0.1/32            scram-sha-256
```

Bien que facultatif dans le cadre du TP, pour restreindre l'accès uniquement au rôle **repli**, il suffit de remplacer la valeur `all` dans le champ dédié aux utilisateurs, par `repli` :

```
host    replication     repli           127.0.0.1/32            scram-sha-256
```

Créer le rôle **repli**, qui sera dédié à la réplication, en lui affectant le mot de passe `confidentiel` :

```
$ createuser --no-superuser --no-createrole --no-createdb --replication -P repli
Enter password for new role:
Enter it again:
```

Configurer ensuite le fichier `.pgpass` de l'utilisateur système `postgres` :

```
$ echo '*:*:*:repli:confidentiel' >> ~/.pgpass
$ chmod 600 ~/.pgpass
```

Il faut recharger la configuration pour qu'elle soit pris en compte par l'instance :

```
$ psql -c 'SELECT pg_reload_conf()'
```

- Créer la première instance secondaire **instance2**, par copie **à chaud** du répertoire de données avec `pg_basebackup` vers `/var/lib/postgresql/16/instance2`.
- Penser à copier les fichiers de configuration
- Penser à modifier le port de cette nouvelle instance avant de la démarrer.

Utiliser `pg_basebackup` pour créer l'instance secondaire :

```
$ pg_basebackup -D /var/lib/postgresql/16/instance2 -P -R -c fast -h 127.0.0.1 \
-U repli

23134/23134 kB (100%), 1/1 tablespace
```

L'option `-R` ou `--write-recovery-conf` de `pg_basebackup` a préparé la configuration de la mise en réplication en créant le fichier `standby.signal` ainsi qu'en configurant `primary_conninfo` dans le fichier `postgresql.auto.conf` (dans les versions antérieures à la 11, il renseignerait `recovery.conf`) :

```
$ cat /var/lib/postgresql/16/instance2/postgresql.auto.conf

primary_conninfo = 'user=repli passfile=''/var/lib/postgresql/.pgpass''
                    channel_binding=prefer host=127.0.0.1 port=5432
                    sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
                    ssl_min_protocol_version=TLSv1.2
                    gssencmode=prefer krbsrvname=postgres gssdelegation=0
                    target_session_attrs=any load_balance_hosts=disable'

$ file /var/lib/postgresql/16/instance2/standby.signal

/var/lib/postgresql/16/instance2/standby.signal: empty
```

Il faut copier le répertoire contenant les fichiers de configuration de l'instance1 :

```
$ cp -r /etc/postgresql/16/instance1 /etc/postgresql/16/instance2
```

Puis, nous devons adapater la configuration présente dans le fichier `postgresql.conf`.

```
$ sed -n -e "s/instance1/instance2/p" -e "s/5432/5433/p" \
/etc/postgresql/16/instance2/postgresql.conf

data_directory = '/var/lib/postgresql/16/instance2'
hba_file = '/etc/postgresql/16/instance2/pg_hba.conf'
ident_file = '/etc/postgresql/16/instance2/pg_ident.conf'
external_pid_file = '/var/run/postgresql/16-instance2.pid'
port = 5433
cluster_name = '16/instance2'
```

La commande suivante permet de rendre les modifications effectives :

```
$ sed -i -e "s/instance1/instance2/" -e "s/5432/5433/" \
/etc/postgresql/16/instance2/postgresql.conf
```

- Démarrer **instance2** et s'assurer que la réplication fonctionne bien avec `ps`.
- Tenter de se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

Il ne reste désormais plus qu'à démarrer l'instance secondaire :

```
# systemctl start  postgresql@16-instance2
```

La commande `ps` suivante permet de voir que les deux serveurs sont lancés :

```
$ ps -o pid,cmd fx
```

La première partie concerne le serveur secondaire :

```
 PID CMD
5321 /usr/lib/postgresql/16/bin/postgres -D /var/lib/postgresql/16/instance2 -c
 ↪  config_file=/etc/postgresql/16/instance2/postgresql.conf
5322  \_ postgres: 16/instance2: checkpointer
5323  \_ postgres: 16/instance2: background writer
5324  \_ postgres: 16/instance2: startup recovering 000000010000000000000003
5325  \_ postgres: 16/instance2: walreceiver streaming 0/3000148
```

La deuxième partie concerne le serveur principal :

```
 PID CMD
4562 /usr/lib/postgresql/16/bin/postgres -D /var/lib/postgresql/16/instance1 -c
 ↪  config_file=/etc/postgresql/16/instance1/postgresql.conf
4563  \_ postgres: 16/instance1: checkpointer
4564  \_ postgres: 16/instance1: background writer
4566  \_ postgres: 16/instance1: walwriter
4567  \_ postgres: 16/instance1: autovacuum launcher
4568  \_ postgres: 16/instance1: logical replication launcher
5326  \_ postgres: 16/instance1: walsender repli 127.0.0.1(41744) streaming 0/3000148
```

Pour différencier les deux instances, il est possible d'identifier le répertoire de données (l'option `-D`), les autres processus sont des fils du processus postmaster. Le paramètre `cluster_name`, déjà configuré sous Debian, permet également de reconnaitre une instance parmi d'autres.

Nous avons bien les deux processus de réplication en flux `wal sender` et `wal receiver`.

Créons quelques données sur le principal et assurons-nous qu'elles soient transmises au secondaire :

```
$ createdb b1

$ psql b1

psql (16.1)
Type "help" for help.

b1=# CREATE TABLE t1(id integer);

CREATE TABLE

b1=# INSERT INTO t1 SELECT generate_series(1, 1000000);

INSERT 0 1000000
```

On constate que le flux a été transmis :

```
b1=# \! ps -o pid,cmd fx | egrep "(startup|walsender|walreceiver)"

 9674  \_ postgres: startup   recovering 000000010000000000000006
 9678  \_ postgres: walreceiver   streaming 0/6D4CD28
 9679  \_ postgres: walsender repli 127.0.0.1(58420) streaming 0/6D4CD28
[...]
```

Essayons de nous connecter au secondaire et d'exécuter quelques requêtes :

```
$ psql -p 5433 b1

psql (16.1)
Type "help" for help.

b1=# SELECT COUNT(*) FROM t1;

  count
---------
 1000000

b1=# CREATE TABLE t2(id integer);

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

On peut se connecter, lire des données, mais pas écrire.

Le comportement est visible dans le log de l'instance secondaire dans le fichier `/var/log/postgresql/postgresql-16-instance2.log` :

```
... LOG:  database system is ready to accept read only connections
```

PostgreSQL indique bien qu'il accepte des connexions en lecture seule.

1.7.2.2 Promotion de l'instance secondaire

> - En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire.

Arrêt de l'instance primaire et vérification de son état :

```
# systemctl stop postgresql@16-instance1
```

```
$ /usr/lib/postgresql/16/bin/pg_controldata -D /var/lib/postgresql/16/instance1 \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               shut down
Latest checkpoint's REDO location:    0/3000148
Latest checkpoint's REDO WAL file:    000000010000000000000003
```

Vérification de l'instance secondaire :

```
$ psql -p 5433 -c 'CHECKPOINT'
```

```
$ /usr/lib/postgresql/16/bin/pg_controldata -D /var/lib/postgresql/16/instance2 \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/3000148
Latest checkpoint's REDO WAL file:    000000010000000000000003
```

L'instance principale est bien arrêtée, l'instance secondaire est bien en `archive recovery` et les deux sont bien synchronisées.

Promotion de l'instance secondaire :

```
$ psql -p 5433 -c 'SELECT pg_promote()'
```

```
 pg_promote
------------
 t
(1 row)
```

> - Tenter de se connecter au serveur secondaire fraîchement promu.
> - Les écritures y sont-elles possibles ?

Connectons-nous à ce nouveau primaire et tentons d'y insérer des données :

```
$ psql -p 5433 b1
psql (16.1)
Type "help" for help.
```

```
b1=# CREATE TABLE t2(id integer);
```

```
CREATE TABLE
```

```
b1=# INSERT INTO t2 SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

Les écritures sont désormais bien possible sur cette instance.

1.7.2.3 Retour à la normale

> - Reconstruire l'instance initiale (`/var/lib/postgresql/16/instance1`) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2** en utilisant `pg_basebackup`.

Afin de rétablir la situation, nous pouvons réintégrer l'ancienne instance primaire en tant que nouveau secondaire. Pour ce faire, nous devons re-synchroniser les données. Utilisons `pg_basebackup` comme précédemment après avoir mis de côté les fichiers de l'ancien primaire :

```
$ mv /var/lib/postgresql/16/instance1 /var/lib/postgresql/16/instance1.old
```

```
$ pg_basebackup -D /var/lib/postgresql/16/instance1 -P -R -c fast \
-h 127.0.0.1 -p 5433 -U repli
```

```
104385/104385 kB (100%), 1/1 tablespace
```

Vérifier la présence du fichier `standby.signal`. Contrôler `postgresql.auto.conf` (qui contient potentiellement deux lignes `primary_conninfo` !). Le fichiers de configuration de l'instance1 n'ayant quant à eux pas été modifiés, il n'est pas nécessaire d'adapter le port d'écoute de l'instance1 par exemple.

```
$ file /var/lib/postgresql/16/instance1/standby.signal
```

```
$ cat /var/lib/postgresql/16/instance1/postgresql.auto.conf
```

```
primary_conninfo = 'user=repli passfile=''/var/lib/postgresql/.pgpass''
                    channel_binding=prefer host=127.0.0.1 port=5433
                    sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
                    ssl_min_protocol_version=TLSv1.2
                    gssencmode=prefer krbsrvname=postgres gssdelegation=0
                    target_session_attrs=any load_balance_hosts=disable'
```

Enfin, démarrer le service :

```
# systemctl start postgresql@16-instance1
```

> - Démarrer cette nouvelle instance.
> - Vérifier que les processus adéquats sont bien présents, et que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

Les processus adéquats sont bien présents :

```
$ ps -o pid,cmd fx | egrep "(startup|walsender|walreceiver)"
```

```
6102  \_ postgres: 16/instance1: startup recovering 000000020000000000000007
6129  \_ postgres: 16/instance1: walreceiver streaming 0/70001F0
6130  \_ postgres: 16/instance2: walsender repli 127.0.0.1(60282) streaming 0/70001F0
```

```
$ psql -p 5432 b1
```

```
psql (16.1)
Type "help" for help.
```

En nous connectant à la nouvelle instance secondaire (port 5432), vérifions que les données précédemment insérées dans la table `t2` sont bien présentes :

```
b1=# SELECT COUNT(*) FROM t2;

  count
---------
 1000000
```

> – Inverser à nouveau les rôles des deux instances afin que **instance2** redevienne l'instance secondaire.

Afin que l'instance 5432 redevienne primaire et celle sur le port 5433 secondaire, on peut ré-appliquer la procédure de promotion vue précédemment dans l'autre sens.

Arrêt de l'instance primaire et vérification de son état :

```
# systemctl stop postgresql@16-instance2

$ /usr/lib/postgresql/16/bin/pg_controldata -D /var/lib/postgresql/16/instance2/ \
| grep -E '(cluster)|(REDO)'

Database cluster state:               shut down
Latest checkpoint's REDO location:    0/70001F0
Latest checkpoint's REDO WAL file:    000000020000000000000007
```

Vérification de l'instance secondaire :

```
$ psql -p 5432 -c 'CHECKPOINT;'
$ /usr/lib/postgresql/16/bin/pg_controldata -D /var/lib/postgresql/16/instance1/ \
| grep -E '(cluster)|(REDO)'

Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/70001F0
Latest checkpoint's REDO WAL file:    000000020000000000000007
```

L'instance principale est bien arrêtée, l'instance secondaire est bien en `archive recovery` et les deux sont bien synchronisées.

Promotion de l'instance secondaire :

```
$ psql -c 'SELECT pg_promote()'

 pg_promote
------------
 t
(1 row)
```

Afin que **instance2** redevienne l'instance secondaire, créer le fichier `standby.signal`, démarrer le service et vérifier que les processus adéquats sont bien présents :

```
$ touch /var/lib/postgresql/16/instance2/standby.signal

#  systemctl start postgresql@16-instance2

$  ps -o pid,cmd fx | egrep "(startup|walsender|walreceiver)"
```

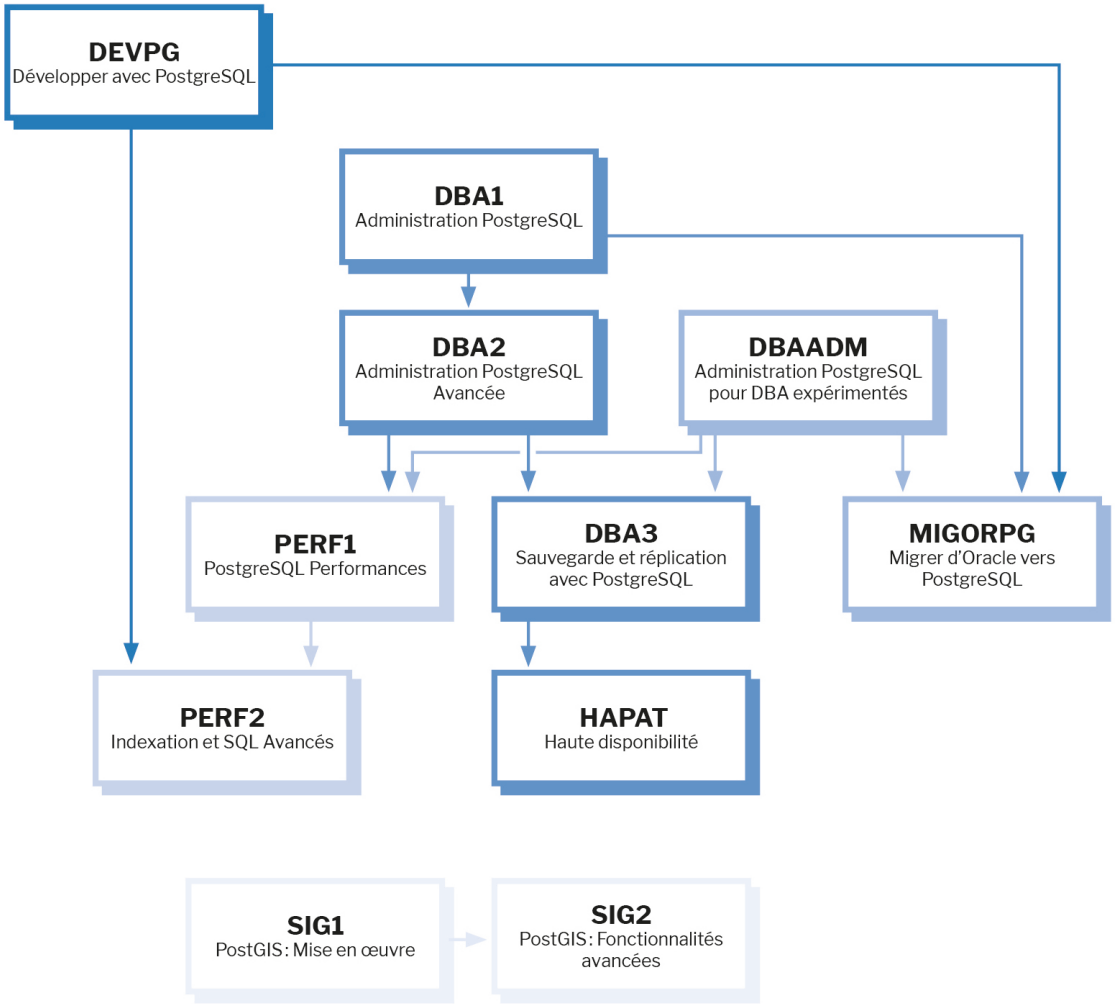
```
6296  \_ postgres: 16/instance2: startup recovering 000000030000000000000007
6299  \_ postgres: 16/instance2: walreceiver streaming 0/7000380
6300  \_ postgres: 16/instance1: walsender repli 127.0.0.1(52208) streaming 0/7000380
```

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.