Module W2A

Réplication Physique : fondamentaux

25.03

Table des matières

Sur ce document . . . 1
Chers lectrices & lecteurs, . . . 1
À propos de DALIBO . . . 1
Remerciements . . . 2
Forme de ce manuel . . . 2
Licence Creative Commons CC-BY-NC-SA . . . 2
Marques déposées . . . 3
Versions de PostgreSQL couvertes . . . 3

1/ Réplication physique : fondamentaux **5**
1.1 Introduction . . . 6
1.1.1 Objectifs . . . 6
1.2 Concepts / principes . . . 7
1.2.1 Principe de la journalisation . . . 7
1.2.2 Principales évolutions de la réplication physique . . . 8
1.2.3 Avantages . . . 9
1.2.4 Inconvénients . . . 10
1.3 Réplication par streaming . . . 11
1.3.1 Mise en place de la réplication par streaming . . . 11
1.3.2 Serveur primaire (1/2) - Configuration . . . 12
1.3.3 Serveur primaire (2/2) - Authentification . . . 13
1.3.4 Serveur secondaire (1/4) - Copie des données . . . 14
1.3.5 Serveur secondaire (2/4) - Fichiers de configuration . . . 15
1.3.6 Serveur secondaire (3/4) - Paramètres . . . 16
1.3.7 Serveur secondaire (4/4) - Démarrage . . . 17
1.3.8 Processus . . . 17
1.4 Promotion . . . 19
1.4.1 Au menu . . . 19
1.4.2 Attention au split-brain ! . . . 19
1.4.3 Vérification avant promotion . . . 20
1.4.4 Promotion du standby : méthode . . . 21
1.4.5 Promotion du standby : déroulement . . . 21
1.4.6 Opérations après promotion du standby . . . 22
1.4.7 Retour à l'état stable . . . 23
1.5 Conclusion . . . 25
1.6 Quiz . . . 26
1.7 Travaux pratiques . . . 27
1.7.1 Sur Rocky Linux 8 ou 9 . . . 27
1.7.2 Sur Debian 12 . . . 28
1.8 Travaux pratiques (solutions) . . . 31
1.8.1 Sur Rocky Linux 8 ou 9 . . . 31
1.8.2 Sur Debian 12 . . . 37

Les formations Dalibo **47**
Cursus des formations 47
Les livres blancs 48
Téléchargement gratuit 48

Sur ce document

Formation	Module W2A
Titre	Réplication Physique : fondamentaux
Révision	25.03
PDF	https://dali.bo/w2a_pdf
EPUB	https://dali.bo/w2a_epub
HTML	https://dali.bo/w2a_html
Slides	https://dali.bo/w2a_slides
TP	https://dali.bo/w2a_tp
TP (solutions)	https://dali.bo/w2a_solutions

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 :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, 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, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, 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

Cette formation est sous licence **CC-BY-NC-SA**[2]. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

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

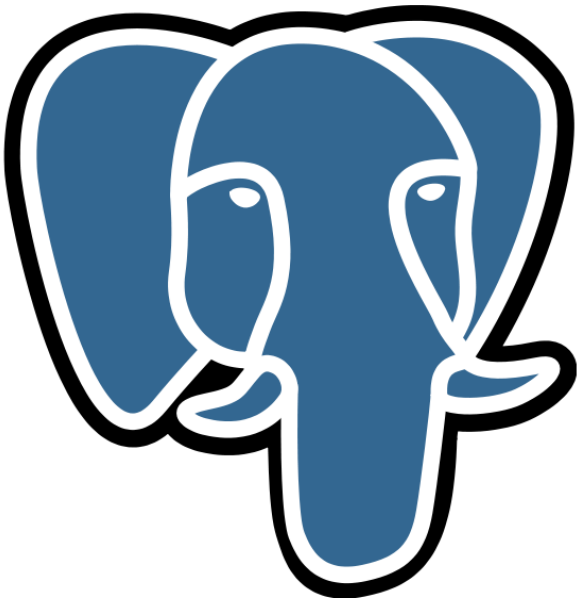
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

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 : fondamentaux



1.1 INTRODUCTION

- Principes
- Mise en place
- Administration

PostgreSQL dispose d'une mécanique de réplication en flux très complète. Ce module permet de comprendre les principes derrière ce type de réplication, sa mise en place et son administration.

1.1.1 Objectifs

- Connaître les avantages et limites de la réplication physique
- Savoir la mettre en place
- Savoir administrer et superviser une solution de réplication physique

1.2 CONCEPTS / PRINCIPES

1.2.1 Principe de la journalisation

- Les journaux de transactions contiennent toutes les modifications
  - utilisation du contenu des journaux
- Le serveur secondaire doit posséder une image des fichiers à un instant t
- La réplication modifiera les fichiers
  - d'après le contenu des journaux suivants

Chaque transaction, implicite ou explicite, réalisant des modifications sur la structure ou les données d'une base est tracée dans les journaux de transactions. Ces derniers contiennent des informations d'assez bas niveau, comme les blocs modifiés sur un fichier suite, par exemple, à un `UPDATE`. La requête elle-même n'apparaît jamais. Les journaux de transactions sont valables pour toutes les bases de données de l'instance.

Les journaux de transactions sont déjà utilisés en cas de crash du serveur. Lors du redémarrage, PostgreSQL rejoue les transactions qui n'auraient pas été synchronisées sur les fichiers de données.

Comme toutes les modifications sont disponibles dans les journaux de transactions et que PostgreSQL sait rejouer les transactions à partir des journaux, il suffit d'archiver les journaux sur une certaine période de temps pour pouvoir les rejouer.

1.2.2 Principales évolutions de la réplication physique

- 8.2 : Réplication par journaux (*log shipping*), *Warm Standby*
- 9.0 : Réplication en *streaming*, *Hot Standby*
- 9.1 à 9.3 : Réplication synchrone, cascade, `pg_basebackup`
- 9.4 : Slots de réplication, délai de réplication, décodage logique
- 9.5 : `pg_rewind`, archivage depuis un serveur secondaire
- 9.6 : Rejeu synchrone
- 10 : Réplication synchrone sur base d'un quorum, slots temporaires
- 12 : Déplacement de la configuration du `recovery.conf` vers le `postgresql.conf`
- 13 : Sécurisation des slots
- 15 : rejeu accéléré

La mise en place de la réplication a été très progressive au fil des versions. Elle pouvait être simpliste au départ, mais elle est à présent au point, et beaucoup plus complète. L'historique permet d'expliquer certaines particularités et complexités.

La version 8.0, en 2005, contenait déjà tout le code qui permet aujourd'hui, après un crash du serveur, de relire les journaux pour rendre à nouveau cohérents les fichiers de données. Pour répliquer une instance, il a suffit d'automatiser l'envoi des journaux vers un serveur secondaire, qui passe son temps à les rejouer, journal après journal (*log shipping*), pour obtenir un serveur prêt à prendre le relai du primaire (*Warm Standby*).

Le serveur secondaire a ensuite été rendu utilisable pour des requêtes en lecture seule (*Hot Standby*).

La réplication a été ensuite améliorée : elle peut se faire en continu (*streaming replication*) et non plus journal par journal, pour réduire le retard du secondaire. Elle peut être synchrone, avec différents niveaux d'arbitrage entre performance et sécurité, et même s'effectuer en cascade, ou avec un délai, et cela en *log shipping* comme en *streaming*.

Puis, l'ajout des slots de réplication a permis au serveur primaire de connaître la position de ses serveurs secondaires, pour savoir quels journaux de transactions sont encore nécessaires.

En parallèle, différents éléments ont été apportés, permettant l'apparition de la réplication logique (qui n'a pas grand-chose à voir avec la réplication physique) en version 10.

L'outil `pg_rewind` a été ajouté pour faciliter la reconstruction d'un ancien serveur primaire devenu secondaire. Il est plus flexible depuis PostgreSQL 13 et peut utiliser un secondaire comme référence depuis la version 14.

La version 10 ajoute la possibilité d'appliquer arbitrairement une réplication synchrone à un sous-ensemble d'un groupe d'instances (_quorum), et non plus juste par ordre de priorité, avec ce paramétrage :

```
synchronous_standby_names = [FIRST]|[ANY] num_sync (node1, node2,...)
```

À partir de PostgreSQL 10, les slots de réplication peuvent être temporaires, et ne durer que le temps de la connexion qui l'a créé.

La version 12 ne change rien sur le fond, mais opère une modification technique lourde : le fichier de paramétrage traditionnel sur le secondaire, `recovery.conf`, disparaît, et ses paramètres sont déplacés dans `postgresql.conf` (ou `postgresql.auto.conf`), ce qui facilite la centralisation de la configuration, et évite d'avoir à redémarrer systématiquement après modification des paramètres concernés. Nous ne parlerons pas ici du paramétrage d'avant la version 12.

La version 13 supprime le plus gros inconvénient des slots de réplication en posant un maximum à la volumétrie qu'ils peuvent conserver (`max_slot_wal_keep_size`).

La version 15 accélère le rejeu du *log shipping*.

Parallèlement à tout cela, les différents outils externes ont également beaucoup progressé, notamment `pg_basebackup`. Par exemple, celui-ci permet une sauvegarde incrémentale à partir de PostgreSQL 17.

1.2.3 Avantages

- Système de rejeu éprouvé
- Mise en place simple
- Pas d'arrêt ou de blocage des utilisateurs
- Réplique tout

Le gros avantage de la réplication par enregistrements de journaux de transactions est sa fiabilité : le système de rejeu qui a permis sa création est un système éprouvé. La mise en place du système complet est simple car son fonctionnement est facile à comprendre. Elle n'implique pas d'arrêt du système, ni de blocage des utilisateurs.

L'autre gros avantage est qu'il réplique tout : modification des données comme évolutions de la structure de la base (DDL), séquences, *large objects*, fonctions… C'est une fonctionnalité que tous les systèmes de réplication logique (notamment par trigger) aimeraient avoir.

1.2.4 Inconvénients

- Réplication de l'instance complète
- Serveur secondaire uniquement en lecture
- Impossible de changer d'architecture
- Même version majeure de PostgreSQL pour tous les serveurs

De manière assez étonnante, l'avantage de tout répliquer est aussi un inconvénient : avec la réplication interne physique de PostgreSQL, il n'est pas possible de ne répliquer qu'une seule base ou que quelques tables.

De même, il n'est pas possible de créer des objets supplémentaires sur le serveur secondaire, comme des index ou des tables de travail, ce qui serait pourtant bien pratique pour de la création de rapports ou pour stocker des résultats intermédiaires de calculs statistiques. Le serveur secondaire est vraiment réservé aux opérations de lecture seule (sauvegardes, répartition de la charge en lecture…) Ces limites ont motivé le développement de la réplication logique pour certains cas d'usage qui ne relèvent pas de la haute disponibilité.

La réplication se passe au niveau du contenu des fichiers et des journaux de transactions. En conséquence, il n'est pas possible d'avoir deux nœuds du système de réplication avec une architecture différente. Par exemple, ils doivent être tous les deux 32 bits ou 64 bits, mais pas un mélange. De même, les deux nœuds doivent être *big endian* ou *little endian*, et doivent aussi être à la même version majeure (pas forcément mineure, ce qui facilite les mises à jours mineures). Pour éviter tout problème de librairie, il est même conseillé d'utiliser des systèmes les plus proches possibles (même distribution de même niveau de mise à jour).

1.3 RÉPLICATION PAR STREAMING

1.3.1 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.

1.3.2 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 indi-

qué 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.

1.3.3 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).

1.3.4 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. Toutefois, à partir de PostgreSQL 17, il permet une sauvegarde incrémentale.

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 les fichiers `postmaster.pid`, `postmaster.opts`, `pg_internal.init`, 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].

1.3.5 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.

1.3.6 Serveur secondaire (3/4) - Paramètres

- `primary_conninfo` (*streaming*) :

```
primary_conninfo = 'user=user_repli host=prod port=5434
 application_name=standby '
```

- Optionnel :
  - `primary_slot_name`
  - `restore_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.

1.3.7 Serveur secondaire (4/4) - 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`.

1.3.8 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
```

1.4 PROMOTION

1.4.1 Au menu

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

1.4.2 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

1.4.3 Vérification avant promotion

- Primaire :

```
# systemctl stop postgresql-17

$ pg_controldata -D /var/lib/pgsql/17/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/17/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. En revanche, **même avec un arrêt propre du primaire**, il peut y avoir perte de données s'il y a un *lag* important entre primaire et secondaire : même si le rejeu va toujours jusqu'au bout avant le changement de *timeline*, les WAL qui n'ont pas pu être récupérés avant la déconnexion, ou après la récupération des archives (si le *log shipping* est en place) sont perdus pour le nouveau primaire.

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.

1.4.4 Promotion du standby : méthode

- Shell :
  - `pg_ctl promote`
- SQL :
  - fonction `pg_promote()`

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 sont à préférer à la méthode historique du fichier de déclenchement (*trigger file*), qui existe encore jusque PostgreSQL 15 inclus. Le paramètre `promote_trigger_file` sur une instance secondaire définit un fichier dont l'apparition provoque la promotion de l'instance. Par sécurité, utiliser un emplacement accessible uniquement aux administrateurs.

1.4.5 Promotion du standby : déroulement

Une promotion déclenche :

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

La promotion se déroule en bonne partie comme un *recovery* après restauration PITR.

Une fois l'instance promue, elle finit de rejouer les derniers journaux de transaction en provenance du serveur principal en sa possession, puis se déconnecte de celui-ci (si l'on est encore connecté en

streaming). Après la déconnexion, si une `restore_command` est configurée, toutes les archives disponibles sont récupérées et rejouées (en général, il n'y a pas d'archive contenant des WAL plus récents que le dernier récupéré en *streaming* ; mais des écritures lourdes et/ou un réseau trop lent peuvent entraîner un retard du *streaming*).

Le dernier journal reçu de l'ancien primaire est souvent incomplet. Il est renommé avec le suffixe `.partial` et archivé. Cela évite un conflit de nom éventuel avec le même fichier issu de l'ancien serveur, qui a pu aussi être archivé, à un point éventuellement postérieur à la divergence.

Ensuite, l'instance choisit une nouvelle *timeline* pour son journal de transactions. Rappelons que la *timeline* est le premier numéro dans le nom du segment (fichier WAL) ; par exemple une timeline 5 pour un fichier nommé `000000050000003200000031`. Le nouveau primaire choisit généralement le numéro suivant celui du primaire (à moins que les archives ne contiennent d'autres *timelines* de numéro supérieur, s'il y a eu plusieurs restaurations et retours en arrière, et il choisit alors le numéro suivant la dernière).

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, des journaux de nom différent permet l'archivage depuis ce primaire sans perturber l'ancien s'il existe encore. Il n'y a plus de fichier 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
```

Un secondaire suit donc par défaut les évolutions de *timeline* de son primaire, tant que celui-ci n'effectue pas de retour en arrière.

L'instance crée un fichier d'historique dans `pg_wal/`, par exemple `00000006.history` pour la nouvelle *timeline* 6. C'est un petit fichier texte qui contient les différentes *timelines* ayant mené à la nouvelle. Ce fichier est immédiatement archivé s'il y a archivage.

Enfin, l'instance autorise les connexions en lecture et en écriture.

1.4.6 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 est conseillé 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.

1.4.7 Retour à l'état stable

Si un *standby* a été momentanément indisponible :

- Rattrapage possible si **tous** les journaux sont disponibles :
  - *streaming* depuis le primaire (slot, `wal_keep_size`)
  - *log shipping* depuis les archives (`restore_command`)
  - ou les deux (si configurés)
- 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 transactions depuis son arrêt soit accessible à ce serveur, **sans exception**.

Si le secondaire ne peut rattraper le flux des journaux du primaire, il peut rester ouvert en lecture, mais il ne rattrapera jamais le primaire. Il est donc irrécupérable et doit être reconstruit par l'une des méthodes précédentes, ce qui peut être long et lourd.

Le secondaire cherche ces journaux par *log shipping* et par *streaming* auprès de son primaire, s'ils sont configurés et fonctionnels. Si une méthode échoue, le secondaire tentera l'autre, alternativement, et indéfiniment jusqu'à obtenir les journaux nécessaires. Si une seule méthode est configurée, le secondaire ne tente bien sûr que celle-ci. Tant qu'une méthode fonctionne, le secondaire l'utilise jusqu'à ce qu'elle tombe en échec.

En cas de réplication par *streaming* :

Le secondaire utilise la `primary_conninfo` pour demander au primaire les journaux à partir de sa propre position dans le flux de journaux, successivement et dans l'ordre. Le primaire ne doit pas avoir

recyclé ou supprimé ces journaux devenus inutiles après leur archivage et un *checkpoint*, ce qui peut ne prendre que quelques minutes.

Si le primaire n'a plus le journal nécessaire, les deux instances afficheront cette erreur :

```
ERROR:  requested WAL segment 000000010000009900000045 has already been removed
```

On peut forcer le primaire à conserver des journaux pour un serveur secondaire :

- La première méthode est simple et consiste à définir un paramètre nommé `wal_keep_size`, pour que le primaire conserve toujours une certaine quantité de journaux, inutiles pour lui, au cas où un secondaire le demande.
- La méthode la plus propre consiste à créer un « slot de réplication » sur le primaire, et dédié à ce secondaire (paramètre `primary_slot` à définir sur celui-ci). Le primaire sait alors en permanence où en est son secondaire, et lui conserve toujours les journaux nécessaires. C'est très sûr, mais peut avoir des inconvénients.

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 ni ne soit corrompu. Souvent, le secondaire utilise le *log shipping* depuis les archives PITR pour rattraper son retard, jusqu'à ce que l'archivage tombe en erreur après épuisement de tous les journaux disponibles ; puis il bascule sur le *streaming* si c'est possible.

1.5 CONCLUSION

- Système de réplication fiable
- Simple à maîtriser et à configurer

La réplication interne à PostgreSQL est le résultat de travaux remontant aussi loin que la version 8.0. Elle est fondée sur des bases solides et saines.

Cette réplication reste fidèle aux principes du moteur de PostgreSQL :

- simple à maîtriser ;
- simple à configurer ;
- fonctionnelle ;
- stable.

1.6 QUIZ

https://dali.bo/w2a_quiz

1.7 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…).

1.7.1 Sur Rocky Linux 8 ou 9

1.7.1.1 Réplication asynchrone en flux avec un seul secondaire

But : Mettre en place une réplication asynchrone en flux.

Ce TP a été mis à jour pour PostgreSQL 17, adapter le numéro de version au besoin.

- Créer l'instance primaire dans `/var/lib/pgsql/17/instance1` (port 5432).

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

- Créer la première instance secondaire **instance2**, port 5433, avec son service systemd, par copie **à chaud** du répertoire de données avec `pg_basebackup` vers `/var/lib/psql/17/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`.
- Se connecter au serveur secondaire.
- Créer quelques tables pour vérifier que les écritures se propagent du primaire au secondaire.

1.7.1.2 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 avec arrêt préalable propre du primaire.

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

1.7.1.3 Retour à la normale

But : Revenir à l'architecture d'origine.

- Reconstruire l'instance initiale (`/var/lib/pgsql/17/instance1`) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2**, en utilisant `pg_basebackup`.

- Démarrer l'**instance1** reconstruite.
- Vérifier 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.

1.7.2 Sur Debian 12

1.7.2.1 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/17/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.

1.7.2.2 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 ?

1.7.2.3 Retour à la normale

But : Revenir à l'architecture d'origine.

- Reconstruire l'instance initiale (`/var/lib/postgresql/17/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.

1.8 TRAVAUX PRATIQUES (SOLUTIONS)

1.8.1 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.
Adapter les numéros de version dans les chemins à la version de PostgreSQL que vous utilisez.

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

Au préalable, arrêter les instances précédemment créées sur le serveur :

```
# systemctl stop postgresql-17
```

1.8.1.1 Réplication asynchrone en flux avec un seul secondaire

- Créer l'instance primaire dans `/var/lib/pgsql/17/instance1` (port 5432), avec le service systemd associé nommé **instance1** (copier et modifier le fichier du service **postgresql-17**).

Le fichier du service est une copie du fichier du service par défaut, en changeant juste le chemin :

```
# cp /lib/systemd/system/postgresql-17.service \
                          /etc/systemd/system/instance1.service

# sed -i "s|/var/lib/pgsql/17/data/|/var/lib/pgsql/17/instance1/|" \
                          /etc/systemd/system/instance1.service
```

ou modifier à la main cette ligne dans `/etc/systemd/system/instance1.service` :

```
Environment=PGDATA=/var/lib/pgsql/17/instance1/
```

Création de la nouvelle instance :

```
# /usr/pgsql-17/bin/postgresql-17-setup initdb instance1
Initializing database ... OK
```

Cette instance doit démarrer. En cas de souci, vérifier qu'il n'existe pas une autre instance qui tourne sur le port 5432.

```
# systemctl start instance1
```

> - Préparer la configuration de la réplication par *streaming* sur ce primaire.
> - L'utilisateur dédié sera nommé **repli**.

Les versions par défaut des paramètres `wal_level` (à `replica`) et `max_wal_sender` (à 10) suffisent, nous n'avons donc pas à modifier `postgresql.conf`.

Nous allons cependant modifier le fichier `/var/lib/pgsql/17/instance1/pg_hba.conf` pour que l'accès en réplication soit autorisé pour l'utilisateur **repli** :

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

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**, port 5433, avec son service systemd, par copie **à chaud** du répertoire de données avec `pg_basebackup` vers `/var/lib/psql/17/instance2`.
> - Penser à modifier le port de cette nouvelle instance avant de la démarrer.

Créons le service systemd de la même manière que ci-dessus :

```
# cp /lib/systemd/system/postgresql-17.service \
                          /etc/systemd/system/instance2.service

# sed -i "s|/var/lib/pgsql/17/data/|/var/lib/pgsql/17/instance2/|" \
                          /etc/systemd/system/instance2.service
```

ou modifier à la main cette ligne dans `/etc/systemd/system/instance2.service` :

```
Environment=PGDATA=/var/lib/pgsql/17/instance2/
```

Utiliser `pg_basebackup` pour copier les données depuis l'instance primaire :

```
$ pg_basebackup -D /var/lib/pgsql/17/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` :

```
$ ls /var/lib/pgsql/17/instance2/standby.signal
```

```
/var/lib/pgsql/17/instance2/standby.signal
```

```
$ cat /var/lib/pgsql/17/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'
```

Noter que ce dernier fichier fait référence au fichier de mot de passe `~/.pgpass`. Sur la même machine il est partagé, mais sur un autre serveur il faudrait le recréer :

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

Puisqu'on travaille sur la même machine, il faut désormais positionner le port d'écoute dans le fichier de configuration, c'est-à-dire `/var/lib/pgsql/17/instance2/postgresql.conf` :

```
port=5433
```

NB : en production, il y aura généralement des machines différentes, donc pas de besoin de changer le port. De plus, la configuration sera soit gérée dans un fichier extérieur au PGDATA (non écrasé à la copie) et appelé depuis `postgresql.conf`, soit adaptée par l'outil qui crée le secondaire (script, playbook…).

> - Démarrer **instance2** et s'assurer que la réplication fonctionne bien avec `ps`.
> - 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 fax |grep postgres
```

```
…
 211738 /usr/pgsql-17/bin/postgres -D /var/lib/pgsql/17/instance1/
 211739  \_ postgres: logger
 211740  \_ postgres: checkpointer
 211741  \_ postgres: background writer
 211743  \_ postgres: walwriter
 211744  \_ postgres: autovacuum launcher
 211745  \_ postgres: archiver last was 000000010000000000000012
 211746  \_ postgres: logical replication launcher
 213912  \_ postgres: walsender repli 127.0.0.1(56444) streaming 0/130001A0
```

```
…
 213902 /usr/pgsql-17/bin/postgres -D /var/lib/pgsql/17/instance2/
 213904  \_ postgres: logger
 213905  \_ postgres: checkpointer
 213906  \_ postgres: background writer
 213907  \_ postgres: startup recovering 000000010000000000000013
 213908  \_ postgres: walreceiver
```

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

Nous avons bien les deux processus de réplication en flux `walsender` et `walreceiver`.

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

```
$ createdb b1
$ psql -d b1
```

```
CREATE TABLE t1(id integer);
-- 1 million de lignes et 35 Mo
INSERT INTO t1 SELECT generate_series(1, 1_000_000);
```

On constate que le flux a été transmis :

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

```
…
 213912  \_ postgres: walsender repli 127.0.0.1(56444) streaming 0/171EDDD8
 213907  \_ postgres: startup recovering 000000010000000000000017
 213908  \_ postgres: walreceiver streaming 0/171EDDD8
```

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

```
$ psql -p 5433 -d b1
```

```
SELECT COUNT(*) FROM t1;
```

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

```
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 traces de l'instance secondaire dans le répertoire `/var/lib/pgsql/17/instance`

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

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

1.8.1.2 Promotion de l'instance secondaire

> - En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire avec arrêt préalable propre du primaire.

Arrêt de l'instance primaire :

```
# systemctl stop instance1
```

Vérification de sa dernière position dans le flux des journaux :

```
$ /usr/pgsql-17/bin/pg_controldata -D /var/lib/pgsql/17/instance1/ \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               shut down
Latest checkpoint's REDO location:    0/16D4E5C8
```

Même vérification pour l'instance secondaire, qui n'est pas arrêtée :

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

```
$ /usr/pgsql-17/bin/pg_controldata -D /var/lib/pgsql/17/instance2/ \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/16D4E5C8
```

L'instance principale est bien arrêtée, l'instance secondaire est bien en *recovery* et elle a bien rattrapé intégralement le flux du primaire.

La promotion de l'instance secondaire se fait soit en ligne de commande :

```
$ /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/17/instance2 promote
```

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

soit en SQL :

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

```
 pg_promote
------------
 t
```

> - 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
```

```
CREATE TABLE t2(id integer);
```

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

```
INSERT 0 1000000
```

Les écritures sont désormais bien possibles sur cette instance.

1.8.1.3 Retour à la normale

> - Détruire puis reconstruire l'instance initiale (`/var/lib/pgsql/17/instance1`) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2**, en utilisant `pg_basebackup`.

Comme la bascule précédente a été contrôlée, il se pourrait que **instance1** puisse être rattachée directement comme secondaire de **instance2**. Mais **instance2** n'a pas forcément gardé assez de journaux. Nous allons donc reconstruire **instance1** de zéro.

Utilisons `pg_basebackup` comme précédemment pour copier les données de **instance2**, après avoir supprimé les fichiers de l'ancien primaire :

```
$ rm -rf -- /var/lib/pgsql/17/instance1
```

```
$ pg_basebackup -D /var/lib/pgsql/17/instance1 -P -R -c fast \
-h 127.0.0.1 -p 5433 -U repli
```

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

Dans `/var/lib/pgsql/17/instance1` :

- créer le fichier `standby.signal` s'il n'existe pas déjà :

```
$ touch /var/lib/pgsql/17/instance1/standby.signal
```

- contrôler `postgresql.auto.conf` (qui contient potentiellement deux lignes `primary_conninfo` !) :

```
$ cat /var/lib/pgsql/17/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/17/instance1/postgresql.conf` :

```
port=5432
```

Enfin, démarrer le service :

```
 # systemctl start instance1
```

> - Démarrer l'**instance1** reconstruite.
> - Vérifier que les données précédemment insérées dans les tables créées plus haut sont bien présentes dans l'instance reconstruite.

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

```
SELECT COUNT(*) FROM t2;
```

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

Les données créées sur **instance2** après sa promotion sont bien présentes.

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

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-17/bin/pg_controldata -D /var/lib/pgsql/17/instance2/ \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               shut down
Latest checkpoint's REDO location:    0/1C000060
```

Vérification de l'instance secondaire :

```
$ psql -p 5432 -c 'CHECKPOINT;'
$ /usr/pgsql-17/bin/pg_controldata -D /var/lib/pgsql/17/instance1/ \
| grep -E '(cluster)|(REDO)'
```

```
Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/1C000060
```

Les deux instances sont bien synchronisées :

Promotion de l'instance secondaire :

```
$ /usr/pgsql-17/bin/pg_ctl -D /var/lib/pgsql/17/instance1 promote
```

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

Afin que **instance2** redevienne l'instance secondaire, il suffit de créer le fichier `standby.signal` et de démarrer le service, puisque la configuration du `primary_conninfo` dans `postgresql.auto.conf` est toujours là.

```
$ touch /var/lib/pgsql/17/instance2/standby.signal
```

```
# systemctl start instance2
```

Vérifier les processus comme précédemment avec :

```
$ ps -o pid,cmd fax |grep postgres
```

Créer des données sur **instance1** (port 5432) doit provoquer leur apparition quasi-immédiate dans **instance2**.

1.8.2 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.

Au préalable, nettoyer les instances qui auraient pu être créées sur le serveur.

```
# pg_dropcluster --stop 17 main
# pg_dropcluster --stop 17 infocentre
# Ceci ne doit rien afficher :
# pg_lsclusters -h
```

1.8.2.1 Réplication asynchrone en flux avec un seul secondaire

- Créer l'instance primaire dans `/var/lib/postgresql/17/instance1` (port 5432) avec `pg_createcluster`.

```
# pg_createcluster 17 instance1
Creating new PostgreSQL cluster 17/instance1 ...
...
Ver Cluster   Port Status Owner    Data directory                  //
17  instance1 5432 down   postgres /var/lib/postgresql/17/instance1 //

Log file
/var/log/postgresql/postgresql-17-instance1.log
```

Le répertoire des données se trouve sous `/var/lib/postgresql/17/instance1`.

Démarrer l'instance, soit avec :

```
# pg_ctlcluster start 17 instance1
```

soit explicitement via systemd :

```
# systemctl start postgresql@17-instance1
```

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

Au sein du fichier `/etc/postgresql/17/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 prise 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/17/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/17/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` :

```
$ cat /var/lib/postgresql/17/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/17/instance2/standby.signal
```

```
/var/lib/postgresql/17/instance2/standby.signal: empty
```

Sous Debian, les fichiers de configuration de l'instance sont sous `/etc`, il faut donc copier ceux d'**instance1** :

```
$ cp -r /etc/postgresql/17/instance1 /etc/postgresql/17/instance2
```

Puis adapter la configuration présente dans le fichier `postgresql.conf` au nouveau nom d'instance :

```
$ sed -i -e "s/instance1/instance2/" -e "s/5432/5433/" \
/etc/postgresql/17/instance2/postgresql.conf
```

Ce qui revient à modifier ces paramètres :

```
data_directory = '/var/lib/postgresql/17/instance2'
hba_file = '/etc/postgresql/17/instance2/pg_hba.conf'
ident_file = '/etc/postgresql/17/instance2/pg_ident.conf'
external_pid_file = '/var/run/postgresql/17-instance2.pid'
port = 5433
cluster_name = '17/instance2'
```

(Une méthode alternative aurait pu être de créer l'**instance2** avec `pg_createcluster`, de l'arrêter et de supprimer ses données. On aurait cependant perdu toute configuration supplémentaire faite sur **instance1**.)

> - 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@17-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/17/bin/postgres -D /var/lib/postgresql/17/instance2 -c
↪  config_file=/etc/postgresql/17/instance2/postgresql.conf
5322  \_ postgres: 17/instance2: checkpointer
5323  \_ postgres: 17/instance2: background writer
5324  \_ postgres: 17/instance2: startup recovering 000000010000000000000003
5325  \_ postgres: 17/instance2: walreceiver streaming 0/3000148
```

La deuxième partie concerne le serveur principal :

```
 PID CMD
4562 /usr/lib/postgresql/17/bin/postgres -D /var/lib/postgresql/17/instance1 -c
↪  config_file=/etc/postgresql/17/instance1/postgresql.conf
4563  \_ postgres: 17/instance1: checkpointer
4564  \_ postgres: 17/instance1: background writer
4566  \_ postgres: 17/instance1: walwriter
4567  \_ postgres: 17/instance1: autovacuum launcher
4568  \_ postgres: 17/instance1: logical replication launcher
5326  \_ postgres: 17/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 `postgres` principal. 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 `walsender` et `walreceiver`.

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

```
$ createdb b1
$ psql -d b1
```

```
CREATE TABLE t1(id integer);
-- 1 million de lignes et 35 Mo
INSERT INTO t1 SELECT generate_series(1, 1_000_000);
```

On constate que le flux a été transmis :

```
$ 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 -d b1
```

```
SELECT COUNT(*) FROM t1;
```

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

```
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-17-instance2.log` :

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

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

1.8.2.2 Promotion de l'instance secondaire

> – En respectant les étapes de vérification de l'état des instances, effectuer une promotion contrôlée de l'instance secondaire avec arrêt préalable propre du primaire.

Arrêt de l'instance primaire :

```
# systemctl stop postgresql@17-instance1
```

Vérification de sa dernière position dans le flux des journaux :

```
$ /usr/lib/postgresql/17/bin/pg_controldata -D /var/lib/postgresql/17/instance1 \
| grep -E '(cluster)|(REDO)'
```

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

Même vérification pour l'instance secondaire, qui n'est pas arrêtée :

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

```
$ /usr/lib/postgresql/17/bin/pg_controldata -D /var/lib/postgresql/17/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 *recovery* et elle a bien rattrapé intégralement le flux du primaire.

La promotion de l'instance secondaire se fait soit en ligne de commande :

```
$ pg_ctlcluster 17 instance2 promote
```

soit en SQL :

```
$ 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
```

```
CREATE TABLE t2(id integer);
INSERT INTO t2 SELECT generate_series(1, 1000000);
```

```
INSERT 0 1000000
```

Les écritures sont désormais bien possibles sur cette instance.

1.8.2.3 Retour à la normale

> - Reconstruire l'instance initiale (`/var/lib/postgresql/17/instance1`) comme nouvelle instance secondaire en repartant d'une copie complète de **instance2**, en utilisant `pg_basebackup`.

Comme la bascule précédente a été contrôlée, il se pourrait que **instance1** puisse être rattachée directement comme secondaire de **instance2**. Mais **instance2** n'a pas forcément gardé assez de journaux. Nous allons donc reconstruire **instance1** de zéro.

Utilisons `pg_basebackup` comme précédemment pour copier les données de **instance2**, après avoir supprimé les fichiers de l'ancien primaire :

```
$ rm -rf -- /var/lib/postgresql/17/instance1
```

```
$ pg_basebackup -D /var/lib/postgresql/17/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.

Dans `/var/lib/postgresql/17/instance1` :

- créer le fichier `standby.signal` s'il n'existe pas déjà :

```
$ touch /var/lib/postgresql/17/instance1/standby.signal
```

- contrôler `postgresql.auto.conf` (qui contient potentiellement deux lignes `primary_conninfo` !) :

```
$ cat /var/lib/postgresql/17/instance1/postgresql.auto.conf
```

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

Enfin, démarrer le service :

```
# systemctl start postgresql@17-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: 17/instance1: startup recovering 000000020000000000000007
6129  \_ postgres: 17/instance1: walreceiver streaming 0/70001F0
6130  \_ postgres: 17/instance2: walsender repli 127.0.0.1(60282) streaming 0/70001F0
```

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

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

```
SELECT COUNT(*) FROM t2;

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

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

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@17-instance2

$ /usr/lib/postgresql/17/bin/pg_controldata -D /var/lib/postgresql/17/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/17/bin/pg_controldata -D /var/lib/postgresql/17/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.

Afin que **instance2** redevienne l'instance secondaire, il suffit de créer le fichier `standby.signal` et de démarrer le service, puisque la configuration du `primary_conninfo` dans `postgresql.auto.conf` est toujours là.

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

 pg_promote
------------
 t
```

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/17/instance2/standby.signal

 #  systemctl start postgresql@17-instance2

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

```
6296  \_ postgres: 17/instance2: startup recovering 000000030000000000000007
6299  \_ postgres: 17/instance2: walreceiver streaming 0/7000380
6300  \_ postgres: 17/instance1: walsender repli 127.0.0.1(52208) streaming 0/7000380
```

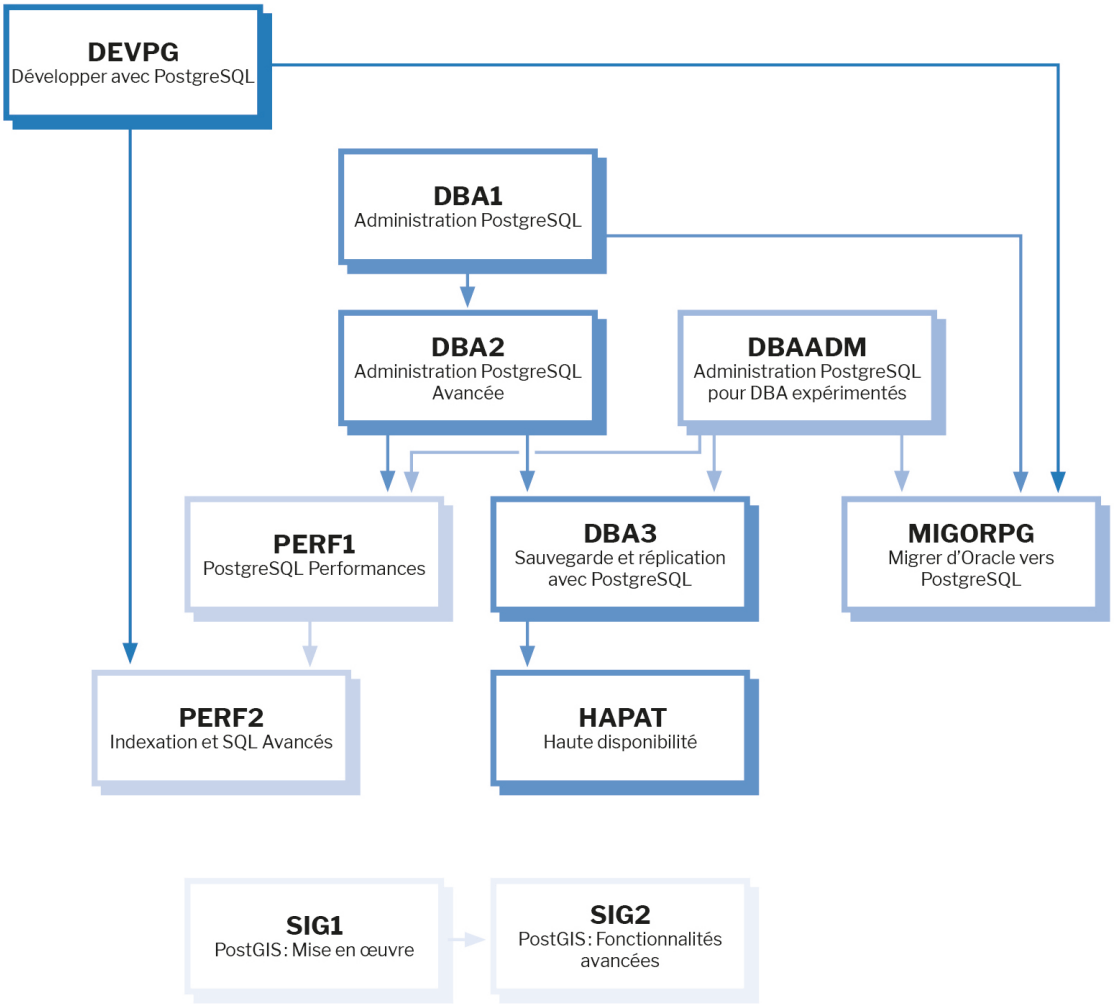
Des données créées sur **instance1** (port 5432) doivent apparaître quasi-immédiatement dans **instance2**.

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.