Module W3

Les outils de réplication

24.04

Table des matières

Sur ce document

Formation	Module W3
Titre	Les outils de réplication
Révision	24.04
PDF	https://dali.bo/w3_pdf
EPUB	https://dali.bo/w3_epub
HTML	https://dali.bo/w3_html
Slides	https://dali.bo/w3_slides
TP	https://dali.bo/w3_tp
TP (solutions)	https://dali.bo/w3_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 :

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

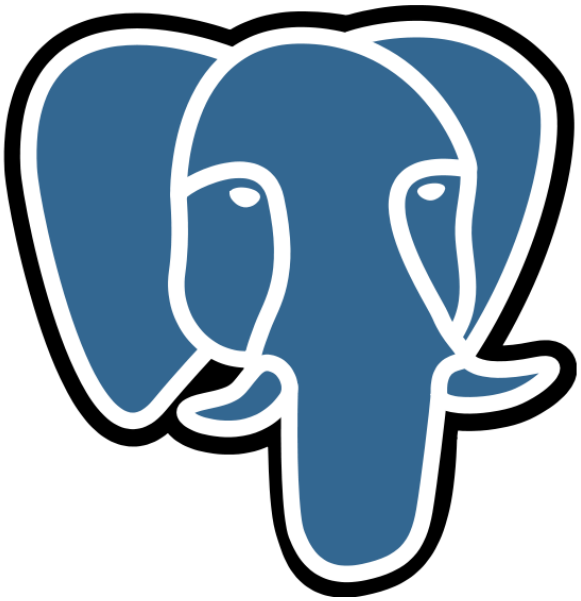
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/ Les outils de réplication



1.1 INTRODUCTION

- Les outils à la rescousse !
  - (Re)construction d'un secondaire
  - Log shipping & PITR
  - Promotion automatique

Nous aborderons dans ce module différents outils externes qui peuvent nous aider à administrer notre cluster de réplication.

1.1.1 Au menu

- (Re)construction d'un secondaire
  - outils de copie
  - pg_rewind
- Log shipping & PITR
  - pgBackRest
  - Barman
- Promotion automatique
  - Patroni
  - repmgr
  - PAF

1.2 (RE)CONSTRUIRE UN SECONDAIRE

- pg_basebackup
- rsync
- outils PITR
- pg_rewind

1.2.1 (Re)construction d'un secondaire : pg_basebackup

- Simple et pratique
- …mais recopie tout !

`pg_basebackup` est un outil éprouvé pour créer une sauvegarde physique, mais aussi un serveur *standby*. Il est basé sur une réplication par *streaming*.

Rappelons ses qualités et défauts dans le cadre de la réplication :

- il sait générer le `postgresql.auto.conf` et le `standby.signal` pour créer un secondaire en *streaming* (option `--write-recovery-conf`) ;
- l'impact sur le réseau peut être limité grâce à l'option `-r` (`--max-rate`) ;
- il peut poser ou utiliser un slot de réplication permanent pour éviter un décrochage dès le démarrage ;
- il peut déplacer des tablespaces à un autre endroit (*mapping*) ;
- les journaux nécessaires peuvent être inclus et leur récupération sécurisée par un slot de réplication temporaire (c'est en général inutile pour (re)construire un secondaire).

Par contre :

- `pg_basebackup` s'attend à créer un *standby* de zéro : le répertoire de données de destination doit être vide, répertoires des tablespaces compris : un ancien secondaire doit donc être détruit (penser à sauver la configuration !) ;
- il copiera à nouveau toutes les données, même si un ancien secondaire ou un ancien primaire presque identique existe ;
- si la connexion est peu stable, le transfert échouera et sera à reprendre de zéro ;
- les données transférées ne sont pas compressées avant la version 15 de PostgreSQL.

(Re)construction :

La sauvegarde générée, et si nécessaire décompressée, ne peut être utilisée directement pour créer une instance secondaire.

Il faut :

- ajouter le fichier `standby.signal` dans le PGDATA ;
- préciser comment récupérer les journaux (`restore_command`) ;
- et/ou paramétrer le *streaming* avec `primary_conninfo` , et éventuellement `primary_slot_name` .

Ces deux derniers points se paramètrent dans `postgresql.conf` ou `postgresql.auto.conf` .

Pour simplifier, `pg_basebackup` peut générer la configuration nécessaire dans le répertoire de la sauvegarde. Par exemple, il ajoutera ceci dans le `postgresql.auto.conf` :

```
primary_conninfo = 'user=postgres passfile=''/var/lib/postgresql/.pgpass''
                    channel_binding=prefer
                    host=localhost port=5432
                    sslmode=prefer sslcompression=0 sslsni=1
                    ssl_min_protocol_version=TLSv1.2
                    gssencmode=prefer krbsrvname=postgres
                    target_session_attrs=any'
```

Cette chaîne réutilise la chaîne fournie en appelant `pg_basebackup` , essaie de prévoir tous les types d'authentification, et positionne nombre de paramètres à leur valeur par défaut. Le plus souvent, utilisateur, port, hôte et `passfile` suffisent.

Si `--slot=nom_du_slot` a été précisé, apparaîtra aussi :

```
primary_slot_name = 'nom_du_slot'
```

Le secondaire possède ainsi immédiatement toutes les informations pour se connecter au primaire. Il est tout de même conseillé de revérifier cette configuration et d'y ajouter la commande de récupération des archives si nécessaire.

1.2.2 (Re)construction d'un secondaire : script rsync

- Intérêts :
  - reprendre des transferts interrompus
  - compression
- Prévoir :
  - `pg_backup_start()` / `pg_backup_stop()`
  - `rsync --whole-file`
  - les tablespaces
  - ne pas tout copier !
  - `postgresql.conf`

Selon les volumes de données mis en jeu, et encore plus avec une liaison instable, il est souvent plus intéressant d'utiliser `rsync`. En effet, `rsync` ne transfère que les fichiers ayant subi une modification. Le transfert sera beaucoup plus rapide s'il existe un secondaire, ou ancien primaire qui a « décroché ». Si le transfert a été coupé, `rsync` permet de le reprendre.

C'est évidemment plus fastidieux qu'un `pg_basebackup` direct, mais peut valoir le coup pour les grosses installations.

Voici un exemple d'utilisation :

```
rsync -av -e 'ssh -o Compression=no' --whole-file --ignore-errors \
      --delete-before --exclude 'lost+found' --exclude 'pg_wal/*' \
      --compress --compress-level=7 \
      --exclude='*.pid' $PRIMARY:$PGDATA/ $PGDATA/
```

Noter que l'on utilise `--whole-file` par précaution pour forcer le transfert entier d'un fichier de données en cas de détection d'une modification. C'est une précaution contre tout risque de corruption (`--inplace` ne transférerait que les blocs modifiés). Les grosses tables sont fractionnées en fichiers de 1 Go, donc elles ne seront pas intégralement retransférées.

Lorsque la connexion utilisée est lente, il est courant de compresser les données pour le transfert (options `-z` / `--compress` et `--compress-level`, de 1 à 9), à ajuster en fonction du CPU disponible.

L'option `--bwlimit` limite au besoin le débit réseau.

Il faudra impérativement encadrer l'appel à `rsync` de `pg_backup_start()` et `pg_backup_stop()`, comme dans une sauvegarde PITR classique.

De nombreux fichiers ne doivent **pas** être copiés. La liste complète figure dans le chapitre sur la

sauvegarde PITR[1] ou la documentation officielle[2]. Les principaux sont `postmaster.pid`, `pg_wal` et ses sous-répertoires, `pg_replslot`, `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` et `pg_subtrans`, `pgsql_tmp*`.

Pour créer le fichier `postgresql.auto.conf` (ou `recovery.conf` si <=v11>), on peut utiliser un fichier modèle tout prêt dans dans le répertoire PGDATA du serveur principal.

La liste des répertoires des tablespaces se trouve dans `$PGDATA/pg_tblspc` sous forme de liens symboliques pointant sur le répertoire du tablespace : on peut alors rendre le script générique pour la synchronisation des tablespaces.

Et il faudra bien tester !

1.2.3 (Re)construction d'un secondaire : outil PITR

- Le plus confortable
- Ne charge pas le primaire
- Mode « delta »

```
pgbackrest --stanza=instance      --type=standby      --delta  \
  --repo1-host=depot --repo1-host-user=postgres --repo1-host-port=22 \
  --pg1-path=/var/lib/postgresql/14/secondaire \
  --recovery-option=primary_conninfo='host=principal port=5433
   ↪  user=replicator' \
  --recovery-option=primary_slot_name='secondaire' \
  --target-timeline=latest \
  restore
```

Si l'on dispose d'un outil PITR, il permet souvent de créer un secondaire, et c'est généralement l'option la plus confortable.

L'exemple ci-dessus utilise pgBackRest pour créer un serveur secondaire dans le répertoire pointé. Les outils concurrents suivent le même principe.

Si le répertoire cible n'est pas vide (secondaire décroché, ancien primaire, restauration échouée…), le paramètre `--delta` permet de ne copier que les différences entre la sauvegarde et le répertoire existant. Cela permet de gagner un temps précieux.

Pour simplifier, une bonne partie de ces options peuvent être définies dans le fichier de configuration local `pgbackrest.conf`.

Les options de *recovery* demandées dans cet exemple apparaîtront dans le fichier `postgresql.auto.conf`, ainsi que la `restore_command` nécessaire pour récupérer les journaux :

[1] https://dali.bo/i2_html
[2] https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP

```
primary_conninfo = 'host=principal port=5433 user=replicator'
primary_slot_name = 'secondaire'
recovery_target_timeline = 'latest'
restore_command = 'pgbackrest --pg1-path=/var/lib/postgresql/14/secondaire
                  --repo1-host=depot --repo1-host-port=22 --repo1-host-user=postgres
                  --stanza=instance archive-get %f "%p"'
```

Un autre intérêt de créer un secondaire depuis une sauvegarde PITR est de ne pas charger le serveur primaire : les fichiers sont copiés depuis le dépôt.

Il y a un petit inconvénient : la sauvegarde peut dater de plusieurs jours, donc il y aura plus de journaux à restaurer qu'en faisant une nouvelle copie du primaire.

1.2.4 pg_rewind

- Outil inclus
- Évite la reconstruction complète malgré une divergence
- Pré-requis :
  - `data-checksums`
  - ou `wal_log_hints = on`
  - tous les WAL depuis la divergence
  - `full_page_writes = on` (défaut)
- Revient au point de divergence

Le cas typique d'utilisation de pg_rewind est de résoudre la divergence entre un ancien primaire et le nouveau primaire après une bascule. Des informations ont été écrites sur l'ancien primaire qui ne sont pas sur le nouveau, et donc empêchent de le raccrocher simplement comme secondaire. Ou encore, un secondaire a été ouvert brièvement en écriture par erreur.

pg_rewind nécessite soit la présence des *checksums* (qui doivent être mis en place à la création de l'instance, ou ajoutés instance arrêtée), soit l'activation du paramètre `wal_log_hints`, et tout cela suffisamment en avance avant la divergence. La volumétrie des journaux de transactions augmentera un peu. `full_page_writes` est aussi nécessaire mais activé par défaut.

Le serveur à récupérer doit être arrêté proprement. Avec PostgreSQL 13, si ce n'est pas le cas, pg_rewind le démarre en mode mono utilisateur puis l'arrête.

Le serveur source (le nouveau primaire) doit être aussi arrêté proprement sur le même serveur, ou accessible par une connexion classique (l'alias `replication` du fichier `pg_hba.conf` ne suffit pas).

L'utilisateur requis pour se connecter à la nouvelle primaire n'est pas forcément superutilisateur. Le droit de se connecter (`LOGIN`) et d'exécuter les fonctions suivantes suffit : `pg_ls_dir()`, `pg_stat_file()`, `pg_read_binary_file(text)` et `pg_read_binary_file(text, bigint, bigint, boolean)`.

Sur l'instance à récupérer, pg_rewind analyse les journaux de transactions depuis le checkpoint précédant la divergence et jusqu'au moment où le serveur a été arrêté pour y récupérer la liste des blocs modifiés. Ces journaux doivent se trouver dans le répertoire `pg_wal` de l'instance. La version 13 permet de s'affranchir de cette limitation en permettant de spécifier une commande de restauration pour récupérer les journaux manquants.

Il récupère ensuite de la nouvelle instance primaire tous les blocs dont la liste a été générée précédemment. Puis, il copie tous les autres fichiers, ce qui inclut ceux des nouvelles relations, les WAL, le contenu du répertoire `pg_xact` et les fichiers de configurations présents dans le répertoire de données. Seuls les fichiers habituellement exclus des sauvegardes ne sont pas récupérés.

Pour finir, il crée un fichier `backup_label` et met à jour le LSN du dernier point de cohérence dans le *control file*.

Une fois l'opération réalisée, l'instance est redémarrée et se met à rejouer tous les journaux depuis le point du dernier checkpoint précédant le point de divergence.

1.3 LOG SHIPPING & PITR

- Utilisation des archives pour :
  - se prémunir du décrochage d'un secondaire
  - une sauvegarde PITR
- Utilisation des sauvegardes PITR pour :
  - resynchroniser un serveur secondaire

Comparé au slot de réplication, le *log shipping* a l'avantage de pouvoir être utilisé pour réaliser des sauvegardes de type PITR. De plus, il est alors possible de se servir de ces sauvegardes pour reconstruire un serveur secondaire plutôt que de réaliser la copie directement depuis l'instance principale.

1.3.1 pgBackRest

- `pgbackrest restore`
  - `--delta`
  - `--type=standby`
  - `--recovery-option`

Il est possible d'utiliser certaines options (en ligne de commande ou fichier de configuration) de l'outil pgBackRest pour resynchroniser une instance secondaire à partir d'une sauvegarde PITR.

Par exemple :

- `--delta` : ne recopiera que les fichiers dont la somme de contrôle (`checksum`) est différente entre la sauvegarde et le répertoire de destination, les fichiers qui ne sont pas présents dans la sauvegarde sont supprimés du répertoire de l'instance ;
- `--type=standby` : crée le fichier `standby.signal` et ajoute la `restore_command` au fichier `postgresql.auto.conf` ;
- `--recovery-option` : permet de spécifier des paramètres particuliers pour la configuration de la restauration (`recovery_target_*`…) ou de la réplication (`primary_conninfo`…).

1.3.2 barman

- `barman recover`
  - `--standby-mode`
  - `--target-*`

Il est possible d'utiliser certaines options de l'outil Barman pour resynchroniser une instance secondaire à partir d'une sauvegarde PITR.

Par exemple :

- `--standby-mode` : crée le fichier `standby.signal` ; la `restore_command` peut également être ajoutée dans `postgresql.auto.conf` ;
- `--target-*` : permettent de spécifier des paramètres particuliers pour la configuration de la restauration (`recovery_target_*`…).

1.4 PROMOTION AUTOMATIQUE

- L'objectif est de minimiser le temps d'interruption du service en cas d'avarie
- Un peu de vocabulaire :
  - *SPOF* : *Single Point Of Failure*
  - *Redondance* : pour éviter les SPOF
  - *Ressources* : éléments gérés par un cluster
  - *Split brain* : deux primaires ! et perte de données
  - *Fencing* : isoler un serveur défaillant
  - *STONITH* : *Shoot The Other Node In The Head* (voir *Fencing*)
  - *Watchdog* : permet à un serveur de s'auto isoler
  - *Quorum* : participe à la résolution des partitions réseau

Pour minimiser le temps d'interruption d'un service, il faut implémenter les éléments nécessaires à la tolérance de panne en se posant la question : que faire si le service n'est plus disponible ?

Une possibilité s'offre naturellement : on prévoit une machine prête à prendre le relais en cas de problème. Lorsqu'on réfléchit aux détails de la mise en place de cette solution, il faut considérer :

- les causes possibles de panne ;
- les actions à prendre face à une panne déterminée et leur automatisation ;
- les situations pouvant corrompre les données.

Les éléments de la plate-forme, aussi matériels que logiciels, dont la défaillance mène à l'indisponibilité du service, sont appelés *SPOF* dans le jargon de la haute disponibilité, ce qui signifie *Single Point Of Failure* ou point individuel de défaillance. L'objectif de la mise en haute disponibilité est d'éliminer ces SPOF. Ce travail concerne le service dans sa globalité : si le serveur applicatif n'est pas hautement-disponible alors que la base de données l'est, le service peut être interrompu car le serveur applicatif est un SPOF. On élimine les différents SPOF possibles par la *redondance*, à la fois matérielle, logicielle mais aussi humaine en évitant qu'une personne détienne toute la connaissance sur un sujet clé.

Par la suite, on discutera de la complexité induite par la mise en haute disponibilité. En effet, dans le cas d'une base de données, éviter les corruptions lors d'événements sur le cluster est primordial.

On peut se trouver dans une situation où les deux machines considèrent qu'elles sont le seul primaire de leur cluster (*Split-Brain*, avec écriture de données sur les deux serveurs simultanément, et la perspective d'une réconciliation délicate) ; ou bien entrent en concurrence pour un accès en écriture sur un même disque partagé.

Pour se protéger d'une partition réseau, on peut utiliser un dispositif implémentant un *Quorum*. Chaque machine se voit allouer un poids qui sera utilisé lors des votes. En cas de coupure réseau, la

partition possédant le plus de votes a le droit de conserver les ressources actives, la ou les autres partitions doivent relâcher les ressources ou les éteindre.

Pour éviter d'avoir des serveurs ou ressources dont l'état est inconnu ou incohérent (serveur injoignable ou ressource qui refuse de s'arrêter), on utilise le *Fencing*. Cette technique permet d'isoler une machine du cluster (*Nœud*) en lui interdisant l'accès à certaines ressources ou en l'arrêtant. Cette deuxième solution est plus connue sous le nom de *STONITH* (*Shoot The Other Node In The Head*). Les techniques pour arriver à ce résultat varient : de la coupure de l'alimentation (PDU, IPMI), à la demande faite au superviseur d'arrêter une machine virtuelle. Une alternative est l'utilisation d'un *Watchdog*. Ce dispositif arrête le serveur s'il n'est pas réarmé à intervalle régulier, ce qui permet à un serveur de s'isoler lui-même.

Pour finir, il est possible d'utiliser un dispositif appelé SBD (*Storage Based Death*) qui combine ces concepts et outils (*fencing* et *watchdog*). Un stockage partagé permet aux serveurs de communiquer pour signaler leur présence ou demander l'arrêt d'un serveur. Si un serveur n'a plus accès au disque partagé, il doit arrêter ses ressources ou s'arrêter si c'est impossible.

Une solution de haute disponibilité robuste combine ces mécanismes pour un maximum de fiabilité.

1.4.1 Patroni

- Outil de HA
- Basé sur un gestionnaire de configuration distribué : etcd, Consul, ZooKeeper…
- Contexte physique, VM ou container
- Spilo : image docker PostgreSQL+Patroni

Patroni[3] est un outil de HA développé par Zalando, nécessitant un gestionnaire de configuration distribué (appelé DCS) tel que ZooKeeper, etcd ou Consul.

Le daemon Patroni fait le lien entre le quorum fourni par le DCS, l'éventuelle utilisation d'un module watchdog, la communication avec les autres nœuds Patroni du cluster et la gestion complète de l'instance PostgreSQL locale : de son démarrage à son arrêt.

Le watchdog est un mécanisme permettant de redémarrer le serveur en cas de défaillance. Il est ici utilisé pour redémarrer si le processus Patroni n'est plus disponible (crash, freeze, bug, etc).

Zalando utilise Patroni principalement dans un environnement conteneurisé. À cet effet, ils ont packagé PostgreSQL et Patroni dans une image docker: Spilo[4].

Cependant, Patroni peut tout à fait être déployé en dehors d'un conteneur, sur une machine physique ou virtuelle.

[3] https://github.com/zalando/patroni
[4] https://github.com/zalando/spilo

1.4.2 repmgr

- Outil spécialisé PostgreSQL
- En pratique, fiable pour 2 nœuds
- Gère automatiquement la bascule en cas de problème
  - *health check*
  - *failover* et *switchover* automatiques
- *Witness*

L'outil repmgr[5] de 2ndQuadrant permet la gestion de la haute disponibilité avec notamment la gestion des opérations de clonage, de promotion d'une instance en primaire et la démotion d'une instance.

L'outil repmgr peut également être en charge de la promotion automatique du nœud secondaire en cas de panne du nœud primaire, cela implique la mise en place d'un serveur supplémentaire par cluster HA (paire primaire/secondaire) appelé témoin (*witness*). Cette machine héberge une instance PostgreSQL dédiée au processus daemon `repmgrd`, processus responsable d'effectuer les contrôles d'accès réguliers à l'instance primaire et de promouvoir le nœud secondaire lorsqu'une panne est détectée et confirmée suite à plusieurs tentatives échouées consécutives.

Afin de faciliter la bascule du trafic sur l'instance secondaire en cas de panne du primaire, l'utilisation d'une adresse IP virtuelle (VIP) est requise. Les applications clientes (hors administration) doivent se connecter directement à la VIP.

1.4.3 Pacemaker

- Solution de Haute Disponibilité généraliste
- Disponible sur les distributions les plus répandues
- Se base sur Corosync, un service de messagerie inter-nœuds
- Permet de surveiller la disponibilité des machines
- Gère le quorum, le fencing, le watchdog et le SBD
- Gère les ressources d'un cluster et leur interdépendance
- Extensible

Pacemaker associe la surveillance de la disponibilité de machines et d'applications. Il offre l'outillage nécessaire pour effectuer les actions suite à une panne. Il s'agit d'une solution de haute disponibilité

[5] https://www.repmgr.org/

extensible avec des scripts.

1.4.4 PAF

- **P**ostgres **A**utomatic **F**ailover
- Ressource Agent pour Pacemaker et Corosync permettant de :
  - détecter un incident
  - relancer l'instance primaire
  - basculer sur un autre nœud en cas d'échec de relance
  - élire le meilleur secondaire (avec le retard le plus faible)
  - basculer les rôles au sein du cluster en primaire et secondaire
- Avec les fonctionnalités offertes par Pacemaker & Corosync :
  - surveillance de la disponibilité du service
  - quorum & Fencing
  - gestion des ressources du cluster

PAF est le fruit de la R&D de Dalibo visant à combler les lacunes des agents existants. Il s'agit d'un produit opensource, disponible sur ce dépôt[6] et qui a rejoint la communauté ClusterLabs qui gère aussi Pacemaker.

[6] https://clusterlabs.github.io/PAF/

1.5 CONCLUSION

De nombreuses applications tierces peuvent nous aider à administrer efficacement un cluster en réplication.

1.5.1 Questions

N'hésitez pas, c'est le moment !

1.6 TRAVAUX PRATIQUES

Pour ces travaux pratiques, procéder à :

- la promotion de l'instance **instance2** ;
- la reconstruction de l'ancienne instance primaire via `pg_rewind` ;
- la sauvegarde de l'instance **instance2** avec pgBackRest ;
- la promotion "par erreur" de l'instance **instance3** ;
- la reconstruction de l'instance **instance3** en se servant des sauvegardes pgBackRest.

1.6.1 (Pré-requis) Environnement

Pour ces TP vous devez avoir 3 instances (1 primaire, 2 secondaires en réplication). Au besoin, repasser en réplication asynchrone en désactivant le paramètre `synchronous_standby_names` de l'instance primaire.

```
$ ps -o pid,cmd fx

  PID CMD
 3205 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance3/
 3207  \_ postgres: logger
 3208  \_ postgres: startup   recovering 000000030000000000000038
 3212  \_ postgres: checkpointer
 3213  \_ postgres: background writer
 3215  \_ postgres: stats collector
 3216  \_ postgres: walreceiver   streaming 0/38000060
 3144 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
 3146  \_ postgres: logger
 3147  \_ postgres: startup   recovering 000000030000000000000038
 3151  \_ postgres: checkpointer
 3152  \_ postgres: background writer
 3154  \_ postgres: stats collector
 3155  \_ postgres: walreceiver   streaming 0/38000060
 2896 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
 2898  \_ postgres: logger
 2900  \_ postgres: checkpointer
 2901  \_ postgres: background writer
 2902  \_ postgres: walwriter
 2903  \_ postgres: autovacuum launcher
 2904  \_ postgres: archiver   last was 000000030000000000000037.00000028.backup
 2905  \_ postgres: stats collector
 2906  \_ postgres: logical replication launcher
 3156  \_ postgres: walsender repli 127.0.0.1(47494) streaming 0/38000060
 3217  \_ postgres: walsender repli 127.0.0.1(47502) streaming 0/38000060

$ psql -x -p 5432 -c "SELECT application_name, state, sent_lsn, write_lsn,
 ↪  flush_lsn, replay_lsn, sync_state FROM pg_stat_replication;"
```

```
-[ RECORD 1 ]----+-----------
application_name | instance2
state            | streaming
sent_lsn         | 0/38000060
write_lsn        | 0/38000060
flush_lsn        | 0/38000060
replay_lsn       | 0/38000060
sync_state       | async
-[ RECORD 2 ]----+-----------
application_name | instance3
state            | streaming
sent_lsn         | 0/38000060
write_lsn        | 0/38000060
flush_lsn        | 0/38000060
replay_lsn       | 0/38000060
sync_state       | async
```

Pour utiliser `pg_rewind`, il faut s'assurer que l'instance a été créée avec l'option `--data-checksums`. Dans le cas contraire, il faut activer `wal_log_hints = on` dans le `postgresql.conf`.

Pour vérifier les valeurs en cours, exécuter en SQL !

```
SHOW data_checksums ;
```

ou utiliser l'outil shell `pg_controldata` :

```
$ /usr/pgsql-14/bin/pg_controldata \
  -D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version:           0
```

Par défaut, une instance est initialisée sans sommes de contrôle. Nous devons donc configurer le paramètre `wal_log_hints` à la valeur `on`. Lançons un *checkpoint* afin de s'assurer que les nouveaux journaux contiendrons les *hints bits* :

```
$ psql -c "CHECKPOINT"
```

Si vous avez suivi correctement les travaux pratiques précédents, vous avez activé les *checksums* et obtenez donc le résultat suivant :

```
$ /usr/pgsql-14/bin/pg_controldata \
  -D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version:           1
```

1.6.2 Promotion d'une instance secondaire

But : Réaliser la promotion d'une instance secondaire.

Contrôler que l'archivage est bien configuré sur l'instance **instance2**.

Arrêter l'instance principale pour promouvoir l'instance **instance2**.

Vérifier le répertoire d'archivage. Que contient-il ?

1.6.3 Suivi de timeline

But : Comprendre la relation entre timeline et réplication.

Nous avons donc :

- l'ancienne instance primaire arrêtée ;
- l'instance **instance2** promue ;
- l'instance **instance3** qui est toujours secondaire de l'instance primaire arrêtée.

Nous allons tenter de raccrocher l'instance **instance3** au nouveau primaire.

- Configurer **instance3** pour qu'il se connecte au **instance2**.

- Observer les traces de l'instance : que se passe-t-il ?

1.6.4 pg_rewind

But : « rembobiner » un ancien primaire avec `pg_rewind`.

Nous allons maintenant expérimenter le cas où l'instance primaire n'a pas été arrêtée avant la bascule. Il peut donc y avoir des transactions dans la *timeline* 2 qui n'existent pas dans la *timeline* 3 ce qui rend impossible la mise en réplication avec le **instance2**.

Comme décrit plus haut, pour utiliser `pg_rewind`, il faut s'assurer que l'instance a été créée avec l'option `--data-checksums` ou que le paramètre `wal_log_hints` est à `on`.

- Redémarrer l'ancienne instance primaire (**instance1**).

- Y créer une table `t7`. Dans quelle *timeline* existe-t-elle ?

- Créer une table `t8` sur **instance2**.

- Arrêter **instance1**.
- La resynchroniser avec **instance2** à l'aide de `pg_rewind`.
- Ne pas oublier de vérifier les fichiers de configuration.
- Configurer cette instance resynchronisée en réplication depuis l'instance **instance2**.
- Ne pas démarrer.

- Démarrer. Observer les traces. Qu'est-il advenu des données insérées avant l'utilisation de `pg_rewind` ?

- Vérifier que **instance2** est bien le serveur principal des deux autres instances.

1.6.5 pgBackRest

But : Utiliser pgBackRest dans le cadre de la réplication.

- Configurer pgBackRest pour sauvegarder votre nouvelle instance principale (**instance2**).

- Effectuer une sauvegarde initiale.
- Insérer des données et vérifier que des archives ont bien été générées.

- Effectuer une promotion « par erreur » du serveur **instance3**, et y insérer des données.

- Resynchroniser l'instance secondaire **instance3** à l'aide de la sauvegarde de pgBackRest.

- Vérifier enfin que la réplication entre **instance3** et **instance2** a bien repris.

1.7 TRAVAUX PRATIQUES (SOLUTIONS)

1.7.1 (Pré-requis) Environnement

Pour ces TP vous devez avoir 3 instances (1 primaire, 2 secondaires en réplication). Au besoin, repasser en réplication asynchrone en désactivant le paramètre `synchronous_standby_names` de l'instance primaire.

```
$ ps -o pid,cmd fx

  PID CMD
 3205 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance3/
 3207  \_ postgres: logger
 3208  \_ postgres: startup   recovering 000000030000000000000038
 3212  \_ postgres: checkpointer
 3213  \_ postgres: background writer
 3215  \_ postgres: stats collector
 3216  \_ postgres: walreceiver   streaming 0/38000060
 3144 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
 3146  \_ postgres: logger
 3147  \_ postgres: startup   recovering 000000030000000000000038
 3151  \_ postgres: checkpointer
 3152  \_ postgres: background writer
 3154  \_ postgres: stats collector
 3155  \_ postgres: walreceiver   streaming 0/38000060
 2896 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance1/
 2898  \_ postgres: logger
 2900  \_ postgres: checkpointer
 2901  \_ postgres: background writer
 2902  \_ postgres: walwriter
 2903  \_ postgres: autovacuum launcher
 2904  \_ postgres: archiver   last was 000000030000000000000037.00000028.backup
 2905  \_ postgres: stats collector
 2906  \_ postgres: logical replication launcher
 3156  \_ postgres: walsender repli 127.0.0.1(47494) streaming 0/38000060
 3217  \_ postgres: walsender repli 127.0.0.1(47502) streaming 0/38000060

$ psql -x -p 5432 -c "SELECT application_name, state, sent_lsn, write_lsn,
 ↪  flush_lsn, replay_lsn, sync_state FROM pg_stat_replication;"

-[ RECORD 1 ]----+-----------
application_name | instance2
state            | streaming
sent_lsn         | 0/38000060
write_lsn        | 0/38000060
flush_lsn        | 0/38000060
replay_lsn       | 0/38000060
sync_state       | async
-[ RECORD 2 ]----+-----------
application_name | instance3
state            | streaming
sent_lsn         | 0/38000060
write_lsn        | 0/38000060
flush_lsn        | 0/38000060
```

```
replay_lsn       | 0/38000060
sync_state       | async
```

Pour utiliser `pg_rewind`, il faut s'assurer que l'instance a été créée avec l'option `--data-checksums`. Dans le cas contraire, il faut activer `wal_log_hints = on` dans le `postgresql.conf`.

Pour vérifier les valeurs en cours, exécuter en SQL !

```
SHOW data_checksums ;
```

ou utiliser l'outil shell `pg_controldata` :

```
$ /usr/pgsql-14/bin/pg_controldata \
  -D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version:           0
```

Par défaut, une instance est initialisée sans sommes de contrôle. Nous devons donc configurer le paramètre `wal_log_hints` à la valeur `on`. Lançons un *checkpoint* afin de s'assurer que les nouveaux journaux contiendrons les *hints bits* :

```
$ psql -c "CHECKPOINT"
```

Si vous avez suivi correctement les travaux pratiques précédents, vous avez activé les *checksums* et obtenez donc le résultat suivant :

```
$ /usr/pgsql-14/bin/pg_controldata \
  -D /var/lib/pgsql/14/instance1/ | grep -E '(checksum)|(sommes de contrôle)'
Data page checksum version:           1
```

1.7.2 Promotion d'une instance secondaire

> Contrôler que l'archivage est bien configuré sur l'instance **instance2**.

```
archive_mode = on
archive_command = 'rsync %p /var/lib/pgsql/14/archives/%f'
```

Avec `archive_mode` à `on`, seule l'instance primaire effectue l'archivage des journaux de transaction.

> Arrêter l'instance principale pour promouvoir l'instance **instance2**.

```
# systemctl stop instance1
```

```
$ psql -p 5433 -c "SELECT pg_promote(true, 60);"
 pg_promote
------------
 t
```

Dans les traces, nous trouvons ceci :

```
LOG:  received promote request
LOG:  redo done at 0/39000028
cp: cannot stat ‘/var/lib/pgsql/14/archives/000000030000000000000039’:
     No such file or directory
```

```
cp: cannot stat ‘/var/lib/pgsql/14/archives/00000004.history’:
    No such file or directory
LOG:  selected new timeline ID: 4
LOG:  archive recovery complete
cp: cannot stat ‘/var/lib/pgsql/14/archives/00000003.history’:
    No such file or directory
LOG:  database system is ready to accept connections
```

Le serveur a bien reçu la demande de promotion, ensuite (si `restore_command` est spécifié) il cherche à rejouer les derniers journaux de transaction depuis les archives. Puis il vérifie la présence d'un fichier `00000003.history` pour déterminer une nouvelle timeline.

Une fois ces opérations effectuées, il détermine la nouvelle *timeline* (ici, 4) et devient accessible en lecture/écriture :

```
$ psql -p 5433 b1

psql (14.1)
Type "help" for help.

b1=# CREATE TABLE t6 (c1 int);
CREATE TABLE
b1=# INSERT INTO t6 VALUES ('1');
INSERT 0 1
```

> Vérifier le répertoire d'archivage. Que contient-il ?

Vu que le paramètre `archive_mode` était à `on`, seul le serveur primaire effectuait l'archivage. En effectuant la promotion de l'instance **instance2**, celle-ci est devenue primaire et archive donc vers le même répertoire !

Il y a bien un archiver process :

```
 3144 /usr/pgsql-14/bin/postmaster -D /var/lib/pgsql/14/instance2/
 3146  \_ postgres: logger
 3151  \_ postgres: checkpointer
 3152  \_ postgres: background writer
 3154  \_ postgres: stats collector
 3309  \_ postgres: walwriter
 3310  \_ postgres: autovacuum launcher
 3311  \_ postgres: archiver   last was 000000030000000000000039.partial
 3312  \_ postgres: logical replication launcher
```

Le répertoire d'archive contient :

```
$ ls -alth /var/lib/pgsql/14/archives/

(...)
-rw-------. 1 postgres postgres 16M Nov 28 16:06 000000030000000000000039.partial
-rw-------. 1 postgres postgres  84 Nov 28 16:06 00000004.history
(...)
```

Le serveur a archivé le dernier journal avec le suffixe `.partial` : ceci afin d'éviter d'écraser un fichier journal si le primaire était toujours actif. Avant la version 9.5, il pouvait y avoir un conflit

entre l'ancien primaire et le secondaire promu archivant au même emplacement. Le fichier `00000003000000000000002F` aurait pu être écrasé. Voir : *Archiving of last segment on timeline after promotion* http://paquier.xyz/postgresql-2/postgres-9-5-feature-highlight-partial-segment-timeline/.

1.7.3 Suivi de timeline

- Configurer **instance3** pour qu'il se connecte au **instance2**.
- Observer les traces de l'instance : que se passe-t-il ?

Dans les traces de **instance3** :

```
2019-11-28 16:13:17.328 CET [3876] FATAL:  could not connect to the primary server:
                could not connect to server: Connection refused
                Is the server running on host "127.0.0.1" and accepting
                TCP/IP connections on port 5432?
```

Le serveur ne sait pas que l'instance **instance2** a été promue, il constate juste que l'instance primaire a été arrêtée.

Modifions le paramètre `primary_conninfo` de l'instance **instance3** pour pointer vers l'instance **instance2** :

```
primary_conninfo = 'host=127.0.0.1 port=5433 user=repli
                    password=repli application_name=instance3'
```

Après redémarrage de l'instance, on peut constater que la modification s'est appliquée sur la *timeline* :

```
(...)
LOG:  database system is ready to accept read only connections
LOG:  started streaming WAL from primary at 0/2F000000 on timeline 4
LOG:  redo starts at 0/390000A0
```

À partir de la version 12, `recovery_target_timeline` est désormais à `latest` par défaut. Précédemment, c'était `current`. Grâce à ce changement, l'instance **instance3** a pu suivre la nouvelle *timeline générée*. Avant la version 12, il aurait fallu modifier ce paramètre.

La table `t6` existe bien et contient bien les enregistrements :

```
$ psql -p 5434 b1 -c "SELECT * FROM t6;"

 c1
----
  1
```

1.7.4 pg_rewind

- Redémarrer l'ancienne instance primaire (**instance1**).

```
$ sudo systemctl start instance1
```

- Y créer une table `t7`. Dans quelle *timeline* existe-t-elle ?

Créons la table `t7` sur **instance1** :

```
$ psql -c "CREATE TABLE t7 (c1 int)" b1
$ psql -c "INSERT INTO t7 VALUES ('1')" b1
```

- Créer une table `t8` sur **instance2**.

Créons la table `t8` sur **instance2** :

```
$ psql -p 5433 -c "CREATE TABLE t8 (c1 int)" b1
$ psql -p 5433 -c "INSERT INTO t8 VALUES ('1')" b1
```

La table `t7` existe dans la timeline 3 mais pas dans la *timeline* 4. Nous allons donc utiliser l'outil `pg_rewind`. Il ne lit et ne copie que les données modifiées. L'outil identifie les blocs correspondants aux transactions « perdues ». Dans notre exemple, la table `t7` n'est présente que sur la nouvelle instance. Ensuite il copie les blocs correspondants sur l'instance cible.

- Arrêter **instance1**.
- La resynchroniser avec **instance2** à l'aide de `pg_rewind`.
- Ne pas oublier de vérifier les fichiers de configuration.
- Configurer cette instance resynchronisée en réplication depuis l'instance **instance2**.
- Ne pas démarrer.

```
$ sudo systemctl stop instance1
```

Maintenant passons à `pg_rewind`. L'option `-n` permet d'effectuer un test sans modification :

```
$ /usr/pgsql-14/bin/pg_rewind -D /var/lib/pgsql/14/instance1/ -n -P \
        --source-server="port=5433 user=postgres"

pg_rewind: connected to server
pg_rewind: servers diverged at WAL location 0/390000A0 on timeline 3
pg_rewind: rewinding from last common checkpoint at 0/39000028 on timeline 3
pg_rewind: reading source file list
pg_rewind: reading target file list
pg_rewind: reading WAL in target
pg_rewind: need to copy 101 MB (total source directory size is 370 MB)
103727/103727 kB (100%) copied
pg_rewind: creating backup label and updating control file
pg_rewind: syncing target data directory
pg_rewind: Done!
```

On relance la commande sans l'option `-n` :

```
$ /usr/pgsql-14/bin/pg_rewind -D /var/lib/pgsql/14/instance1/ -P \
        --source-server="port=5433 user=postgres"

pg_rewind: connected to server
pg_rewind: servers diverged at WAL location 0/390000A0 on timeline 3
pg_rewind: rewinding from last common checkpoint at 0/39000028 on timeline 3
pg_rewind: reading source file list
pg_rewind: reading target file list
pg_rewind: reading WAL in target
pg_rewind: need to copy 101 MB (total source directory size is 370 MB)
103735/103735 kB (100%) copied
pg_rewind: creating backup label and updating control file
pg_rewind: syncing target data directory
pg_rewind: Done!
```

On constate qu'il a juste été nécessaire de copier 101 Mo au lieu des 370 Mo de l'instance complète.

`pg_rewind` se connecte sur le serveur source et identifie le point de divergence. Ensuite, il liste les fichiers à copier ou supprimer. Puis il copie les blocs modifiés. Enfin il met à jour le fichier `backup_label` et le fichier `pg_control`.

Les fichiers `postgresql.conf` et `postgresql.auto.conf` provenant du **instance2**, il faut modifier le `port` ainsi que le `primary_conninfo` :

```
### postgresql.conf
port = 5432

### postgresql.auto.conf
primary_conninfo = 'user=repli passfile=''/var/lib/pgsql/.pgpass''
                    host=127.0.0.1 port=5433 application_name=instance1'
```

Créer le fichier `standby.signal` :

```
$ touch /var/lib/pgsql/14/instance1/standby.signal
```

Et démarrer l'instance :

```
$ sudo systemctl start instance1
```

> – Démarrer. Observer les traces. Qu'est-il advenu des données insérées avant l'utilisation de `pg_rewind` ?

Dans les traces :

```
LOG:  database system was interrupted while in recovery at log time ...
HINT:  If this has occurred more than once some data might be corrupted and
you might need to choose an earlier recovery target.
cp: cannot stat ‘/var/lib/pgsql/14/archives/00000005.history’: No such
file or directory
LOG:  entering standby mode
LOG:  restored log file "00000004.history" from archive
cp: cannot stat ‘/var/lib/pgsql/14/archives/000000040000000000000039’: No
```

```
such file or directory
cp: cannot stat ‘/var/lib/pgsql/14/archives/00000003.history’: No such
file or directory
LOG:  redo starts at 0/390000A0
LOG:  started streaming WAL from primary at 0/39000000 on timeline 4
LOG:  consistent recovery state reached at 0/3904A8E0
LOG:  database system is ready to accept read only connections
```

La table `t7` a bien disparu de l'instance resynchronisée :

```
$ psql -p 5432 -c "SELECT * FROM t7;" b1

ERROR:  relation "t7" does not exist
```

- Vérifier que **instance2** est bien le serveur principal des deux autres instances.

```
$ psql -x -p 5433 -c "SELECT * FROM pg_stat_replication;"

-[ RECORD 1 ]----+------------------------------
pid              | 3610
usesysid         | 16384
usename          | repli
application_name | instance3
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 49942
backend_start    | ...
backend_xmin     |
state            | streaming
sent_lsn         | 0/3904B628
write_lsn        | 0/3904B628
flush_lsn        | 0/3904B628
replay_lsn       | 0/3904B628
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | ...
-[ RECORD 2 ]----+------------------------------
pid              | 3797
usesysid         | 16384
usename          | repli
application_name | instance1
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 49966
backend_start    | ...
backend_xmin     |
state            | streaming
sent_lsn         | 0/3904B628
write_lsn        | 0/3904B628
flush_lsn        | 0/3904B628
replay_lsn       | 0/3904B628
write_lag        |
flush_lag        |
```

```
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | ...
```

1.7.5 pgBackRest

- Configurer pgBackRest pour sauvegarder votre nouvelle instance principale (**instance2**).

Commencer par modifier :

```
archive_command = 'pgbackrest --stanza=instance_dev archive-push %p'
```

Ensuite, recharger la configuration :

```
$ sudo systemctl reload instance2
```

Modifier ensuite la configuration de pgBackRest dans `/etc/pgbackrest.conf` :

```
[global]
start-fast=y
log-level-console=info
process-max=1
repo1-path=/var/lib/pgsql/14/backups
repo1-retention-full=1

[instance_dev]
pg1-path=/var/lib/pgsql/14/instance2
pg1-port=5433
```

Vérifier que le répertoire où les archives et les sauvegardes seront stockées est bien vide (`/var/lib/pgsql/14/backups/`).

Initialiser le dépôt de pgBackRest et vérifier que l'archivage est fonctionnel :

```
$ pgbackrest --stanza=instance_dev stanza-create
P00   INFO: stanza-create command begin 2.20: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance2 --pg1-port=5433
--repo1-path=/var/lib/pgsql/14/backups --stanza=instance_dev
P00   INFO: stanza-create command end: completed successfully (463ms)

$ pgbackrest --stanza=instance_dev check
P00   INFO: check command begin 2.20: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance2 --pg1-port=5433
--repo1-path=/var/lib/pgsql/14/backups --stanza=instance_dev
P00   INFO: WAL segment 000000040000000000000039 successfully archived to
'/var/lib/pgsql/14/backups/archive/instance_dev/14-1/0000000400000000/
000000040000000000000039-9b054265590ac75cd70ba3ed5b1f47c113a69b84.gz'
P00   INFO: check command end: completed successfully
```

- Effectuer une sauvegarde initiale.
- Insérer des données et vérifier que des archives ont bien été générées.

Pour une première sauvegarde :

```
$ pgbackrest --stanza=instance_dev --type=full backup |grep P00
P00   INFO: backup command begin 2.20: --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance2 --pg1-port=5433 --process-max=1
--repo1-path=/var/lib/pgsql/14/backups --repo1-retention-full=1
--stanza=instance_dev --start-fast --type=full
P00   INFO: execute non-exclusive pg_start_backup() with label "...": backup
begins after the requested immediate checkpoint completes
P00   INFO: backup start archive = 00000004000000000000003B, lsn = 0/3B000060
P00   INFO: full backup size = 274.2MB
P00   INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments
to archive
P00   INFO: backup stop archive = 00000004000000000000003B, lsn = 0/3B000138
P00   INFO: new backup label = 20200110-081833F
P00   INFO: backup command end: completed successfully
P00   INFO: expire command begin 2.20: --log-level-console=info
--repo1-path=/var/lib/pgsql/14/backups --repo1-retention-full=1
--stanza=instance_dev
P00   INFO: expire command end: completed successfully
```

Insérer des données :

```
$ psql -p 5433 -c "CREATE TABLE t9(id int);" b1
CREATE TABLE

$ psql -p 5433 -c "INSERT INTO t9 SELECT generate_series(0,1000000);" b1
INSERT 0 1000001
```

Forcer la génération d'un nouveau WAL et vérifier que des archives ont bien été générées :

```
$ psql -p 5433 -c "select pg_switch_wal();"
 pg_switch_wal
---------------
 0/3FD474E8
(1 row)

$  psql -p 5433 -c "select last_archived_wal from pg_stat_archiver;"
    last_archived_wal
--------------------------
 00000004000000000000003F
(1 row)
```

- Effectuer une promotion « par erreur » du serveur **instance3**, et y insérer des données.

```
$ psql -p 5434 -c "SELECT pg_promote(true, 60);"
 pg_promote
------------
 t

$ psql -p 5434 -c "CREATE TABLE t10(id int);" b1
CREATE TABLE
```

- Resynchroniser l'instance secondaire **instance3** à l'aide de la sauvegarde de pgBackRest.

De nouvelles données ayant été écrites, nous devons désormais reconstruire l'instance **instance3** pour pouvoir la remettre en réplication avec **instance2**. Pour ce faire, nous pourrions utiliser `pg_rewind` comme précédemment. Nous allons ici utiliser les sauvegardes pgBackRest pour reconstruire l'instance.

La sauvegarde de l'instance primaire peut être restaurée sur la seconde instance avec les options `--delta` et `--type=standby` pour écraser les fichiers erronés et ajouter le fichier `standby.signal` dans le répertoire de données. pgBackRest se charge de configurer les options de réplication dans le fichier `postgresql.auto.conf`.

```
$ sudo systemctl stop instance3
$ pgbackrest restore --stanza=instance_dev --delta --type=standby
--recovery-option="primary_conninfo=user=repli passfile=''/var/lib/pgsql/.pgpass''
host=127.0.0.1 port=5433 application_name=instance3"
--pg1-path=/var/lib/pgsql/14/instance3 |grep P00

P00   INFO: restore command begin 2.20: --delta --log-level-console=info
--pg1-path=/var/lib/pgsql/14/instance3 --process-max=1
--recovery-option="primary_conninfo=user=repli passfile=''/var/lib/pgsql/.pgpass''
host=127.0.0.1 port=5433 application_name=instance3"
--repo1-path=/var/lib/pgsql/14/backups --stanza=instance_dev --type=standby
P00   INFO: restore backup set ...
P00   INFO: remap data directory to '/var/lib/pgsql/14/instance3'
P00   INFO: remove invalid files/links/paths from '/var/lib/pgsql/14/instance3'
P00   INFO: write updated /var/lib/pgsql/14/instance3/postgresql.auto.conf
P00   INFO: restore global/pg_control (performed last to ensure aborted restores
            cannot be started)
P00   INFO: restore command end: completed successfully
```

- Vérifier enfin que la réplication entre **instance3** et **instance2** a bien repris.

Modifier le port de l'instance **instance3** avant de la démarrer et vérifier que la connexion de réplication est bien fonctionnelle :

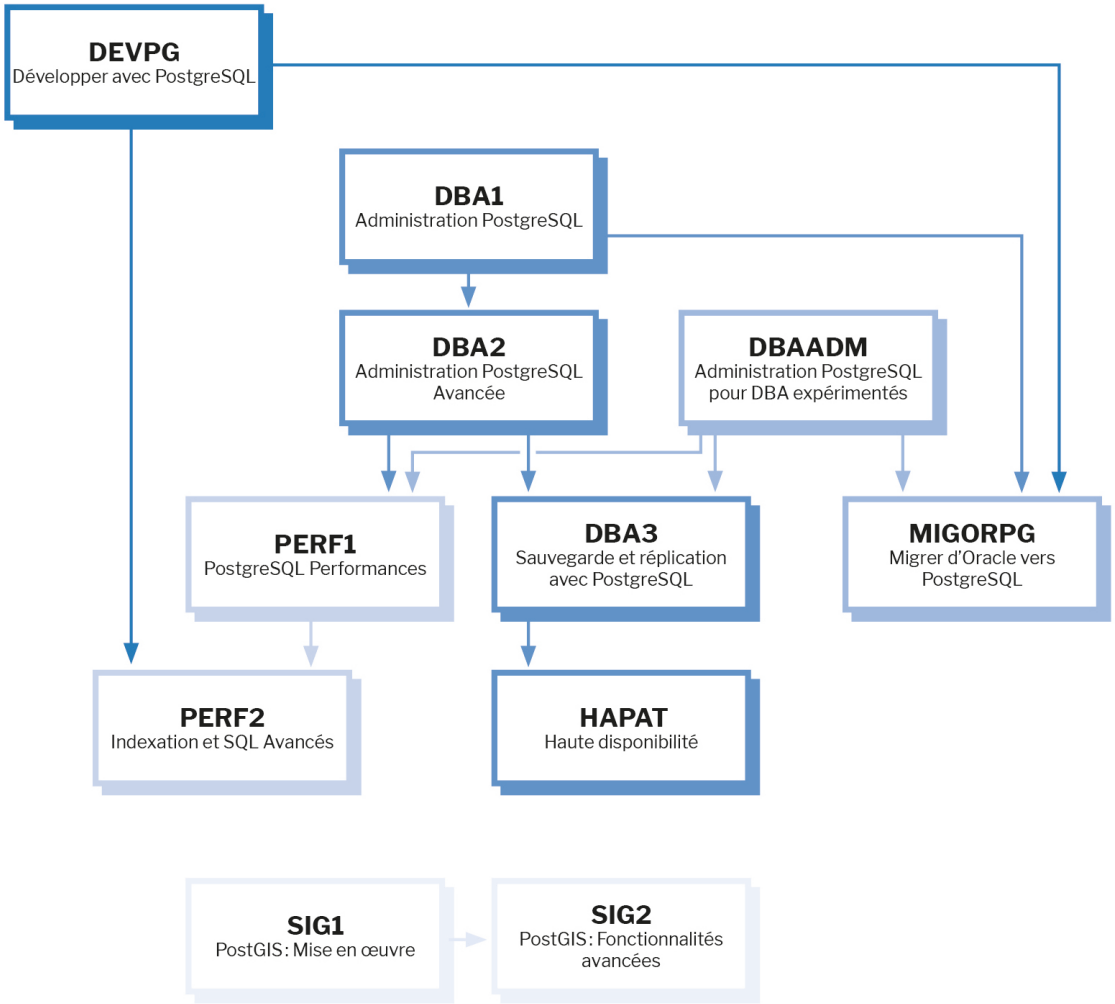
```
port=5434

$ sudo systemctl start instance3
$ psql -x -p 5433 -c "SELECT application_name, state, sync_state FROM
 ↪  pg_stat_replication WHERE application_name='instance3';"

-[ RECORD 1 ]----+-------------------------------
application_name | instance3
state            | streaming
sync_state       | async
```

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.