Formation DBA3

PostgreSQL Sauvegardes et Réplication

24.04

Table des matières

Sur ce document

Formation	Formation DBA3
Titre	PostgreSQL Sauvegardes et Réplication
Révision	24.04
ISBN	978-2-38168-110-8
PDF	https://dali.bo/dba3_pdf
EPUB	https://dali.bo/dba3_epub
HTML	https://dali.bo/dba3_html
Slides	https://dali.bo/dba3_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

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/ PostgreSQL : Politique de sauvegarde

Photo de l'incendie du datacenter OVHcloud à Strasbourg du 10 mars 2021 fournie gracieusement par l'ITBR67[1].

Cet incendie a provoqué de nombreux arrêts et pertes de données dans toute la France et ailleurs[2].

[1] https://itbr67.fr/
[2] https://fr.wikipedia.org/wiki/Incendie_du_centre_de_donn%C3%A9es_d%27OVHcloud_%C3%A0_Strasbourg

1.1 INTRODUCTION

- Le pire peut arriver
- Politique de sauvegarde

1.1.1 Au menu

- Objectifs
- Approche
- Points d'attention

1.2 DÉFINIR UNE POLITIQUE DE SAUVEGARDE

- Pourquoi établir une politique ?
- Que sauvegarder ?
- À quelle fréquence sauvegarder les données ?
- Quels supports ?
- Quels outils ?
- Vérifier la restauration des sauvegardes

Afin d'assurer la sécurité des données, il est nécessaire de faire des sauvegardes régulières.

Ces sauvegardes vont servir, en cas de problème, à restaurer les bases de données dans un état le plus proche possible du moment où le problème est survenu.

Cependant, le jour où une restauration sera nécessaire, il est possible que la personne qui a mis en place les sauvegardes ne soit pas présente. C'est pour cela qu'il est essentiel d'écrire et de maintenir un document qui indique la mise en place de la sauvegarde et qui détaille comment restaurer une sauvegarde.

En effet, suivant les besoins, les outils pour sauvegarder, le contenu de la sauvegarde, sa fréquence ne seront pas les mêmes.

Par exemple, il n'est pas toujours nécessaire de tout sauvegarder. Une base de données peut contenir des données de travail, temporaires et/ou faciles à reconstruire, stockées dans des tables standards. Il est également possible d'avoir une base dédiée pour stocker ce genre d'objets. Pour diminuer le temps de sauvegarde (et du coup de restauration), il est possible de sauvegarder partiellement son serveur pour ne conserver que les données importantes.

La fréquence peut aussi varier. Un utilisateur peut disposer d'un serveur PostgreSQL pour un entrepôt de données, serveur qu'il n'alimente qu'une fois par semaine. Dans ce cas, il est inutile de sauvegarder tous les jours. Une sauvegarde après chaque alimentation (donc chaque semaine) est suffisante. En fait, il faut déterminer la fréquence de sauvegarde des données selon :

- le volume de données à sauvegarder et/ou restaurer ;
- la criticité des données ;
- la quantité de données qu'il est « acceptable » de perdre en cas de problème.

Le support de sauvegarde est lui aussi très important. Il est possible de sauvegarder les données sur un disque réseau (à travers SMB/CIFS ou NFS), sur des disques locaux dédiés, sur des bandes ou tout autre support adapté. Dans tous les cas, il est fortement déconseillé de stocker les sauvegardes sur les disques utilisés par la base de données.

Ce document doit aussi indiquer comment effectuer la restauration. Si la sauvegarde est composée de plusieurs fichiers, l'ordre de restauration des fichiers peut être essentiel. De plus, savoir où se trouvent les sauvegardes permet de gagner un temps important, qui évitera une immobilisation trop longue.

De même, vérifier la restauration des sauvegardes de façon régulière est une précaution très utile.

1.2.1 Objectifs

- Sécuriser les données
- Mettre à jour le moteur de données
- Dupliquer une base de données de production
- Archiver les données

L'objectif essentiel de la sauvegarde est la sécurisation des données. Autrement dit, l'utilisateur cherche à se protéger d'une panne matérielle ou d'une erreur humaine (un utilisateur qui supprimerait des données essentielles). La sauvegarde permet de restaurer les données perdues. Mais ce n'est pas le seul objectif d'une sauvegarde.

Une sauvegarde peut aussi servir à dupliquer une base de données sur un serveur de test ou de préproduction. Elle permet aussi d'archiver des tables. Cela se voit surtout dans le cadre des tables partitionnées où l'archivage de la table la plus ancienne permet ensuite sa suppression de la base pour gagner en espace disque.

Un autre cas d'utilisation de la sauvegarde est la mise à jour majeure de versions PostgreSQL. Il s'agit de la solution historique de mise à jour (export/import). Historique, mais pas obsolète.

1.2.2 Différentes approches

- Sauvegarde à froid des fichiers (ou physique)
- Sauvegarde à chaud en SQL (ou logique)
- Sauvegarde à chaud des fichiers (PITR)

À ces différents objectifs vont correspondre différentes approches de la sauvegarde.

La sauvegarde au niveau système de fichiers permet de conserver une image cohérente de l'intégralité des répertoires de données d'une instance arrêtée. C'est la sauvegarde à froid. Cependant, l'utilisation d'outils de snapshots pour effectuer les sauvegardes peut accélérer considérablement les temps de sauvegarde des bases de données, et donc diminuer d'autant le temps d'immobilisation du système.

La sauvegarde logique permet de créer un fichier texte de commandes SQL ou un fichier binaire contenant le schéma et les données de la base de données.

La sauvegarde à chaud des fichiers est possible avec le *Point In Time Recovery*.

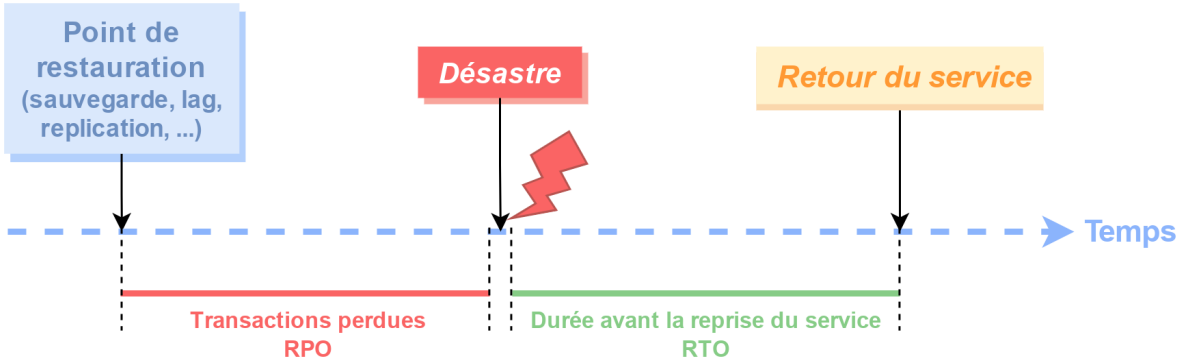
Suivant les prérequis et les limitations de chaque méthode, il est fort possible qu'une seule de ces solutions soit utilisable. Par exemple :

- si le serveur ne peut pas être arrêté, la sauvegarde à froid est exclue d'office ;
- si la base de données est très volumineuse, la sauvegarde logique devient très longue ;
- si l'espace disque est limité et que l'instance génère beaucoup de journaux de transactions, la sauvegarde PITR sera difficile à mettre en place.

1.2.3 RTO/RPO

La politique de sauvegarde découle du :

- **RPO** (*Recovery Point Objective*) : Perte de Données Maximale Admissible
  - faible ou importante ?
- **RTO** (*Recovery Time Objective*) : Durée Maximale d'Interruption Admissible
  - courte ou longue ?



Le RPO et RTO sont deux concepts déterminants dans le choix des politiques de sauvegardes.

La **RPO** (ou PDMA[3]) est la perte de données maximale admissible, ou quantité de données que l'on peut tolérer de perdre lors d'un sinistre majeur, souvent exprimée en heures ou minutes.

Pour un système mis à jour épisodiquement ou avec des données non critiques, ou facilement récupérables, le RPO peut être important (par exemple une journée). Peuvent alors s'envisager des solutions comme :

- les sauvegardes logiques (dump) ;
- les sauvegardes des fichiers à froid.

[3] https://fr.wikipedia.org/wiki/Perte_de_donn%C3%A9es_maximale_admissible

Dans beaucoup de cas, la perte de données admissible est très faible (heures, quelques minutes), voire nulle. Il faudra s'orienter vers des solutions de type :

- sauvegarde à chaud ;
- sauvegarde d'instantané à un point donnée dans le temps (PITR) ;
- réplication asynchrone, voire synchrone.

La **RTO** (ou DMIA[4]) est la durée maximale d'interruption du service.

Dans beaucoup de cas, les utilisateurs peuvent tolérer une indisponibilité de plusieurs heures, voire jours. La durée de reprise du service n'est alors pas critique, on peut utiliser des solutions simples comme :

- la restauration des fichiers ;
- la restauration d'une sauvegarde logique (dump).

Si elle est plus courte, le service doit très vite remonter. Cela nécessite des procédures avec un minimum d'acteurs et de manipulation :

- réplication ;
- solutions HA (Haute Disponibilité).

Plus le besoin en RTO/RPO sera court, plus les solutions seront complexes à mettre en œuvre — et chères. Inversement, pour des données non critiques, un RTO/RPO long permet d'utiliser des solutions simples et peu coûteuses.

1.2.4 Industrialisation

- Évaluer les coûts humains et matériels
- Intégrer les méthodes de sauvegardes avec le reste du SI
  - sauvegarde sur bande centrale
  - supervision
  - plan de continuité et de reprise d'activité

Les moyens nécessaires pour la mise en place, le maintien et l'intégration de la sauvegarde dans le SI ont un coût financier qui apporte une contrainte supplémentaire sur la politique de sauvegarde.

Du point de vue matériel, il faut disposer principalement d'un volume de stockage qui peut devenir conséquent. Cela dépend de la volumétrie à sauvegarder, il faut considérer les besoins suivants :

[4] https://fr.wikipedia.org/wiki/Dur%C3%A9e_maximale_d%27interruption_admissible#RTO

- Stocker plusieurs sauvegardes. Même avec une rétention d'une sauvegarde, il faut pouvoir stocker la suivante durant sa création : il vaut mieux purger les anciennes sauvegardes une fois qu'on est sûr que la sauvegarde s'est correctement déroulée.
- Avoir suffisamment de place pour restaurer sans avoir besoin de supprimer la base ou l'instance en production. Un tel espace de travail est également intéressant pour réaliser des restaurations partielles. Cet espace peut être mutualisé. On peut utiliser également le serveur de préproduction s'il dispose de la place suffisante.

Avec une rétention d'une sauvegarde unique, il est bon de prévoir 3 fois la taille de la base ou de l'instance. Pour une faible volumétrie, cela ne pose pas de problèmes, mais quand la volumétrie devient de l'ordre du téraoctet, les coûts augmentent significativement.

L'autre poste de coût est la mise en place de la sauvegarde. Une équipe de DBA peut tout à fait décider de créer ses propres scripts de sauvegarde et restauration, pour diverses raisons, notamment :

- maîtrise complète de la sauvegarde, maintien plus aisé du code ;
- intégration avec les moyens de sauvegardes communs au SI (bandes, externalisation…) ;
- adaptation au PRA/PCA plus fine.

Enfin, le dernier poste de coût est la maintenance, à la fois des scripts et par le test régulier de la restauration.

1.2.5 Documentation

- Documenter les éléments clés de la politique :
  - perte de données
  - rétention
  - temps de référence
- Documenter les processus de sauvegarde et restauration
- Imposer des révisions régulières des procédures

Comme pour n'importe quelle procédure, il est impératif de documenter la politique de sauvegarde, les procédures de sauvegarde et de restauration ainsi que les scripts.

Au strict minimum, la documentation doit permettre à un DBA non familier de l'environnement de comprendre la sauvegarde, retrouver les fichiers et restaurer les données le cas échéant, le plus rapidement possible et sans laisser de doute. En effet, en cas d'avarie nécessitant une restauration, le service aux utilisateurs finaux est généralement coupé, ce qui génère un climat de pression propice aux erreurs qui ne fait qu'empirer la situation.

L'idéal est de réviser la documentation régulièrement en accompagnant ces révisions de tests de restauration : avoir un ordre de grandeur de la durée d'une restauration est primordial. On demandera toujours au DBA qui restaure une base ou une instance combien de temps cela va prendre.

1.2.6 Règle 3-2-1

- 3 exemplaires des données
- 2 sur différents médias
- 1 hors site (et hors ligne)
- Un RAID n'est pas une sauvegarde !
- Le *cloud* n'est pas une solution magique !

L'un des points les plus importants à prendre en compte est l'endroit où sont stockés les fichiers des sauvegardes. Laisser les sauvegardes sur la même machine n'est pas suffisant : si une défaillance matérielle se produisait, les sauvegardes pourraient être perdues en même temps que l'instance sauvegardée, rendant ainsi la restauration impossible.

Il est conseillé de suivre au moins la règle 3-2-1. Les données elles-mêmes sont le premier exemplaire.

Les deux copies doivent se trouver sur des supports physiques différents (et de préférence sur un autre serveur) pour parer à la destruction du support original (notamment une perte de disques durs). La première copie peut être à proximité pour faciliter une restauration.

Des disques en RAID ne sont pas une sauvegarde ! Ils peuvent parer à la défaillance d'un disque, pas à une fausse manipulation (`rm -rf /` ou `TRUNCATE` malheureux). La perte de la carte contrôleur peut entraîner la perte de toute la grappe. Un conseil courant est d'ailleurs de choisir des disques de séries différentes pour éviter des défaillances simultanées.

Le troisième exemplaire doit se trouver à un autre endroit, pour parer aux scénarios les plus catastrophiques (cambriolage, incendie…). Selon la criticité, le délai nécessaire pour remonter rapidement un système fonctionnel, et le budget, ce troisième exemplaire peut être une copie manuelle sur un disque externe stocké dans un coffre, ou une infrastructure répliquée complète avec sa copie des sauvegardes à l'autre bout de la ville, voire du pays.

Pour limiter la consommation d'espace disque des copies multiples, les durées de rétention peuvent différer. La dernière sauvegarde peut résider sur la machine, les 5 dernières sur un serveur distant, des bandes déposées dans un site sécurisé tous les mois.

Stocker vos données dans le *cloud* n'est pas une solution miracle. Un datacenter peut brûler entièrement[5]. Dans la formule choisie, il faut donc bien vérifier que le fournisseur sauvegarde les données

[5] https://dali.bo/20210310-ovh-incendie-strasbourg

sur un autre site. Mais on a aussi vu un hébergeur perdre *toutes* les données de ses clients à cause d'un ransomware[6].

N'hésitez donc pas à faire vous-même une copie de vos données.
Il ne faut pas non plus négliger le risque d'une attaque (piratage, malveillance ou *ransomware*…), qui s'en prendra à toute sauvegarde accessible en ligne. Une copie physique hors ligne est donc chaudement recommandée pour les données les plus critiques.

1.2.7 Autres points d'attention

- Sauvegarder les fichiers de configuration
- **Tester la restauration**
  - De nombreuses catastrophes auraient pu être évitées avec un test
  - Estimation de la durée

La sauvegarde ne concerne pas uniquement les données. Il est également fortement conseillé de sauvegarder les fichiers de configuration du serveur et les scripts d'administration. Ils sont parfois générés par l'outil d'industrialisation, gérés par un outil externe (Patroni…), ou versionnés ailleurs. L'idéal est de les copier avec les sauvegardes. On peut également déléguer cette tâche à une sauvegarde au niveau système, vu que ce sont de simples fichiers. Les principaux fichiers de PostgreSQL à prendre en compte sont : `postgresql.conf`, `postgresql.auto.conf`, `pg_hba.conf`, `pg_ident.conf`. Ces fichiers ont parfois des clauses d'inclusion pour en appeler d'autres. Cette liste n'est en aucun cas exhaustive.

Il s'agit donc de recenser l'ensemble des fichiers et scripts nécessaires si l'on désirait recréer le serveur depuis zéro.

Enfin, même si les sauvegardes se déroulent correctement, il est indispensable de tester si elles se restaurent sans erreur. Une erreur de copie lors de l'externalisation peut, par exemple, rendre la sauvegarde inutilisable.

[6] https://dcmag.fr/cloudnordic-le-datacenter-qui-a-perdu-toutes-les-donnees-de-ses-clients-lors-dune-attaque-par-ransomware/

Just that backup tapes are seen to move, backup scripts are run for a lengthy period of time, should not be construed as verifying that data backups are properly being performed.
Que l'on voit bouger les bandes de sauvegardes, ou que les scripts de sauvegarde fonctionnent pendant une longue période, ne doit pas être interprété comme une validation que les sauvegardes sont faites.

Le test de restauration permet de vérifier l'ensemble de la procédure : ensemble des objets sauvegardés, intégrité de la copie, commandes à passer pour une restauration complète (en cas de stress, vous en oublierez probablement une partie). De plus, cela permet d'estimer la durée nécessaire à la restauration.

Nous rencontrons régulièrement en clientèle des scripts de sauvegarde qui ne fonctionnent pas, et jamais testés. Vous trouverez sur Internet de nombreuses histoires de catastrophes qui auraient été évitées par un simple test. Entre mille autres :

- une base disparaît à l'arrêt et ses sauvegardes sont vides[7] : erreur de manipulation, script ne vérifiant pas qu'il a bien fait sa tâche, monitoring défaillant ;
- perte de 6 heures de données chez Gitlab en 2017[8] : procédures de sauvegarde et de réplication incomplètes, complexes et peu claires, outils mal choisis ou mal connus, sauvegardes vides et jamais testées, distraction d'un opérateur — Gitlab a le mérite d'avoir tout soigneusement documenté[9].

Restaurer régulièrement les bases de test ou de préproduction à partir des sauvegardes de la production est une bonne idée. Cela vous évitera de découvrir la procédure dans l'urgence, le stress, voire la panique, alors que vous serez harcelé par de nombreux utilisateurs ou clients bloqués.

[7] http://www.pgdba.org/post/restoring_data/#a-database-horror-story
[8] https://about.gitlab.com/2017/02/01/gitlab-dot-com-database-incident/
[9] https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/

1.3 CONCLUSION

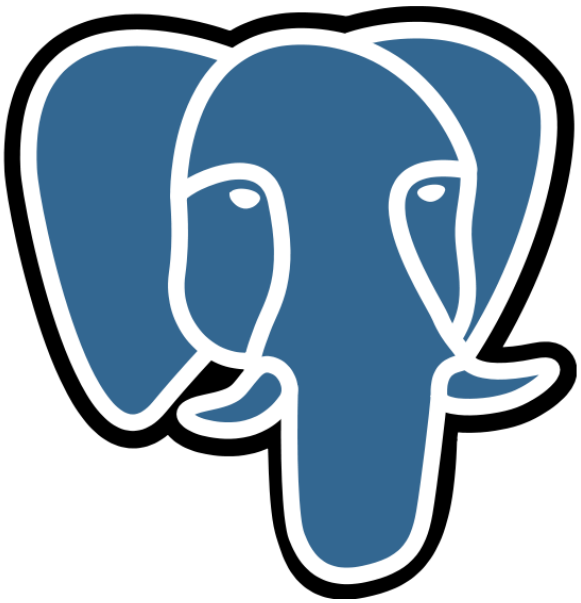
- Les techniques de sauvegarde de PostgreSQL sont :
  - complémentaires
  - automatisables
- La maîtrise de ces techniques est indispensable pour assurer un service fiable.
- **Testez vos sauvegardes !**

L'écosystème de PostgreSQL offre tout le nécessaire pour effectuer des sauvegardes fiables. Le plan de sauvegarde doit être fait sérieusement, et les sauvegardes testées. Cela a un coût, mais un désastre détruisant toutes vos données sera incommensurablement plus ruineux.

1.4 QUIZ

https://dali.bo/i0_quiz

2/ Sauvegarde physique à chaud et PITR



2.1 INTRODUCTION

- Sauvegarde traditionnelle
  - sauvegarde `pg_dump` à chaud
  - sauvegarde des fichiers à froid
- Insuffisant pour les grosses bases
  - long à sauvegarder
  - encore plus long à restaurer
- Perte de données potentiellement importante
  - car impossible de réaliser fréquemment une sauvegarde
- Une solution : la sauvegarde PITR

La sauvegarde traditionnelle, qu'elle soit logique ou physique à froid, répond à beaucoup de besoins. Cependant, ce type de sauvegarde montre de plus en plus ses faiblesses pour les gros volumes : la sauvegarde est longue à réaliser et encore plus longue à restaurer. Et plus une sauvegarde met du temps, moins fréquemment on l'exécute. La fenêtre de perte de données devient plus importante.

PostgreSQL propose une solution à ce problème avec la sauvegarde physique à chaud. On peut l'utiliser comme un simple mode de sauvegarde supplémentaire, mais elle permet bien d'autres possibilités, d'où le nom de PITR (*Point In Time Recovery*).

2.1.1 Au menu

- Mettre en place la sauvegarde PITR
  - sauvegarde : manuelle, ou avec `pg_basebackup`
  - archivage : manuel, ou avec `pg_receivewal`
- Restaurer une sauvegarde PITR
- Des outils

Ce module fait le tour de la sauvegarde PITR, de la mise en place de l'archivage (manuelle ou avec l'outil `pg_receivewal`) à la sauvegarde des fichiers (en manuel, ou avec l'outil `pg_basebackup`). Il

discute aussi de la restauration d'une telle sauvegarde. Nous évoquerons très rapidement quelques outils externes pour faciliter ces sauvegardes.

2.2 PITR

- *Point In Time Recovery*
- À chaud
- En continu
- Cohérente

PITR est l'acronyme de *Point In Time Recovery*, autrement dit restauration à un point dans le temps.

C'est une sauvegarde à chaud et surtout en continu. Là où une sauvegarde logique du type `pg_dump` se fait au mieux une fois toutes les 24 h, la sauvegarde PITR se fait en continu grâce à l'archivage des journaux de transactions. De ce fait, ce type de sauvegarde diminue très fortement la fenêtre de perte de données.

Bien qu'elle se fasse à chaud, la sauvegarde est cohérente.

2.2.1 Principes

- Les journaux de transactions contiennent toutes les modifications
- Il faut les archiver
  - …et avoir une image des fichiers à un instant t
- La restauration se fait en restaurant cette image
  - …et en rejouant les journaux
  - dans l'ordre
  - entièrement
  - ou partiellement (*ie* jusqu'à un certain moment)

Quand une transaction est validée, les données à écrire dans les fichiers de données sont d'abord écrites dans un journal de transactions. Ces journaux décrivent donc toutes les modifications survenant sur les fichiers de données, que ce soit les objets utilisateurs comme les objets systèmes. Pour reconstruire un système, il suffit donc d'avoir ces journaux et d'avoir un état des fichiers du répertoire des données à un instant t. Toutes les actions effectuées après cet instant t pourront être rejouées en demandant à PostgreSQL d'appliquer les actions contenues dans les journaux. Les opérations stockées dans les journaux correspondent à des modifications physiques de fichiers, il faut donc partir d'une sauvegarde au niveau du système de fichier, un export avec `pg_dump` n'est pas utilisable.

Il est donc nécessaire de conserver ces journaux de transactions. Or PostgreSQL les recycle dès qu'il n'en a plus besoin. La solution est de demander au moteur de les archiver ailleurs avant ce recyclage. On doit aussi disposer de l'ensemble des fichiers qui composent le répertoire des données (incluant les tablespaces si ces derniers sont utilisés).

La restauration a besoin des journaux de transactions archivés. Il ne sera pas possible de restaurer et éventuellement revenir à un point donné avec la sauvegarde seule. En revanche, une fois la sauvegarde des fichiers restaurée et la configuration réalisée pour rejouer les journaux archivés, il sera possible de les rejouer tous ou seulement une partie d'entre eux (en s'arrêtant à un certain moment). Ils doivent impérativement être rejoués dans l'ordre de leur écriture (et donc de leur nom).

2.2.2 Avantages

- Sauvegarde à chaud
- Rejeu d'un grand nombre de journaux
- Moins de perte de données

Tout le travail est réalisé à chaud, que ce soit l'archivage des journaux ou la sauvegarde des fichiers de la base. En effet, il importe peu que les fichiers de données soient modifiés pendant la sauvegarde car les journaux de transactions archivés permettront de corriger toute incohérence par leur application.

Il est possible de rejouer un très grand nombre de journaux (une journée, une semaine, un mois, etc.). Évidemment, plus il y a de journaux à appliquer, plus cela prendra du temps. Mais il n'y a pas de limite au nombre de journaux à rejouer.

Dernier avantage, c'est le système de sauvegarde qui occasionnera le moins de perte de données. Généralement, une sauvegarde `pg_dump` s'exécute toutes les nuits, disons à 3 h du matin. Supposons qu'un gros problème survienne à midi. S'il faut restaurer la dernière sauvegarde, la perte de données sera de 9 h. Le volume maximum de données perdu correspond à l'espacement des sauvegardes. Avec l'archivage continu des journaux de transactions, la fenêtre de perte de données va être fortement réduite. Plus l'activité est intense, plus la fenêtre de temps sera petite : il faut changer de fichier de journal pour que le journal précédent soit archivé et les fichiers de journaux sont de taille fixe.

Pour les systèmes n'ayant pas une grosse activité, il est aussi possible de forcer un changement de journal à intervalle régulier, ce qui a pour effet de forcer son archivage, et donc dans les faits de pouvoir s'assurer une perte correspondant au maximum à cet intervalle.

2.2.3 Inconvénients

- Sauvegarde de l'instance complète
- Nécessite un grand espace de stockage (données + journaux)
- Risque d'accumulation des journaux
  - dans `pg_wal` en cas d'échec d'archivage… avec arrêt si il est plein !
  - dans le dépôt d'archivage si échec des sauvegardes
- Restauration de l'instance complète
- Impossible de changer d'architecture (même OS conseillé)
- Plus complexe

Certains inconvénients viennent directement du fait qu'on copie les fichiers : sauvegarde et restauration complète (impossible de ne restaurer qu'une seule base ou que quelques tables), restauration sur la même architecture (32/64 bits, *little/big endian*). Il est même fortement conseillé de restaurer dans la même version du même système d'exploitation, sous peine de devoir réindexer l'instance (différence de définition des locales notamment).

Elle nécessite en plus un plus grand espace de stockage car il faut sauvegarder les fichiers (dont les index) ainsi que les journaux de transactions sur une certaine période, ce qui peut être volumineux (en tout cas beaucoup plus que des `pg_dump`).

En cas de problème dans l'archivage et selon la méthode choisie, l'instance ne voudra pas effacer les journaux non archivés. Il y a donc un risque d'accumulation de ceux-ci. Il faudra donc surveiller la taille du `pg_wal`. En cas de saturation, PostgreSQL s'arrête !

Enfin, la sauvegarde PITR est plus complexe à mettre en place qu'une sauvegarde `pg_dump`. Elle nécessite plus d'étapes, une réflexion sur l'architecture à mettre en œuvre et une meilleure compréhension des mécanismes internes à PostgreSQL pour en avoir la maîtrise.

2.3 COPIE PHYSIQUE À CHAUD PONCTUELLE AVEC PG_BASEBACKUP

- Réalise les différentes étapes d'une sauvegarde
  - via 1 ou 2 connexions de réplication + slots de réplication
  - base backup + journaux nécessaires
- Copie intégrale,
  - image de la base à la **fin** du backup
- Pas de copie incrémentale
- Configuration : *streaming* (rôle, droits, slots)

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

Description :

`pg_basebackup` est un produit qui a beaucoup évolué dans les dernières versions de PostgreSQL. De plus, le paramétrage par défaut le rend immédiatement utilisable.

Il permet de réaliser toute la sauvegarde de l'instance, à distance, via deux connexions de réplication : une pour les données, une pour les journaux de transactions qui sont générés pendant la copie. Sa compression permet d'éviter une durée de transfert ou une place disque occupée trop importante. Cela a évidemment un coût, notamment au niveau CPU, sur le serveur ou sur le client suivant le besoin. Il est simple à mettre en place et à utiliser, et permet d'éviter de nombreuses étapes que nous verrons par la suite.

Par contre, il ne permet pas de réaliser une sauvegarde incrémentale, et ne permet pas de continuer à archiver les journaux, contrairement aux outils de PITR classiques. Cependant, ceux-ci peuvent l'utiliser pour réaliser la première copie des fichiers d'une instance.

Mise en place :

`pg_basebackup` nécessite des connexions de réplication. Il peut utiliser un slot de réplication, une technique qui fiabilise la sauvegarde ou la réplication en indiquant à l'instance quels journaux elle doit conserver. Par défaut, tout est en place pour une connexion en local :

```
wal_level = replica
max_wal_senders = 10
max_replication_slots = 10
```

Ensuite, il faut configurer le fichier `pg_hba.conf` pour accepter la connexion du serveur où est exécutée `pg_basebackup`. Dans notre cas, il s'agit du même serveur avec un utilisateur dédié :

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

Enfin, il faut créer un utilisateur dédié à la réplication (ici `sauve`) qui sera le rôle créant la connexion et lui attribuer un mot de passe :

```
CREATE ROLE sauve LOGIN REPLICATION;
\password sauve
```

Dans un but d'automatisation, le mot de passe finira souvent dans un fichier `.pgpass` ou équivalent.

Il ne reste plus qu'à :

- lancer `pg_basebackup`, ici en lui demandant une archive au format `tar` ;
- archiver les journaux en utilisant une connexion de réplication par *streaming* ;
- forcer le *checkpoint*.

Cela donne la commande suivante, ici pour une sauvegarde en local :

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/

644320/644320 kB (100%), 1/1 tablespace
```

Le résultat est ici un ensemble des deux archives : les journaux sont à part et devront être dépaquetés dans le `pg_wal` à la restauration.

```
$ ls -l /var/lib/postgresql/backups/
total 4163772
-rw------- 1 postgres postgres 659785216 Oct  9 11:37 base.tar
-rw------- 1 postgres postgres  16780288 Oct  9 11:37 pg_wal.tar
```

La cible doit être vide. En cas d'arrêt avant la fin, il faudra tout recommencer de zéro, c'est une limite de l'outil.

Restauration :

Pour restaurer, il suffit de remplacer le PGDATA corrompu par le contenu de l'archive, ou de créer une nouvelle instance et de remplacer son PGDATA par cette sauvegarde. Au démarrage, l'instance repérera qu'elle est une sauvegarde restaurée et réappliquera les journaux. L'instance contiendra les données telles qu'elles étaient à la **fin** du `pg_basebackup`.

Noter que les fichiers de configuration ne sont PAS inclus s'ils ne sont pas dans le PGDATA, notamment sur Debian et ses versions dérivées.

Différences entre les versions :

Un slot temporaire sera créé par défaut pour garantir que le serveur gardera les journaux jusqu'à leur copie intégrale.

À partir de la version 13, la commande `pg_basebackup` crée un fichier manifeste contenant la liste des fichiers sauvegardés, leur taille et une somme de contrôle. Cela permet de vérifier la sauvegarde avec

l'outil `pg_verifybackup` (ce dernier ne fonctionne hélas que sur une sauvegarde au format `plain`, ou décompressée).

Lisez bien la documentation de pg_basebackup[1] pour votre version précise de PostgreSQL, des options ont changé de nom au fil des versions.

Même avec un serveur un peu ancien, il est possible d'installer un `pg_basebackup` récent, en installant les outils clients de la dernière version de PostgreSQL.

L'outil est développé plus en détail dans notre module I4[2].

[1] https://docs.postgresql.fr/current/app-pgbasebackup.html
[2] https://dali.bo/i4_html

2.4 SAUVEGARDE PITR

2 étapes :

- Archivage des journaux de transactions
  - archivage interne
  - ou outil `pg_receivewal`
- Sauvegarde des fichiers
  - `pg_basebackup`
  - ou manuellement (outils de copie classiques)

Même si la mise en place est plus complexe qu'un `pg_dump`, la sauvegarde PITR demande peu d'étapes. La première chose à faire est de mettre en place l'archivage des journaux de transactions. Un choix est à faire entre un archivage classique et l'utilisation de l'outil `pg_receivewal`.

Lorsque cette étape est réalisée (et fonctionnelle), il est possible de passer à la seconde : la sauvegarde des fichiers. Là-aussi, il y a différentes possibilités : soit manuellement, soit `pg_basebackup`, soit son propre script ou un outil extérieur.

2.4.1 Méthodes d'archivage

- Deux méthodes :
  - processus interne `archiver`
  - outil `pg_receivewal` (flux de réplication)

La méthode historique est la méthode utilisant le processus `archiver`. Ce processus fonctionne sur le serveur à sauvegarder et est de la responsabilité du serveur PostgreSQL. Seule sa (bonne) configuration incombe au DBA.

Une autre méthode existe : `pg_receivewal`. Cet outil livré aussi avec PostgreSQL se comporte comme un serveur secondaire. Il reconstitue les journaux de transactions à partir du flux de réplication.

Chaque solution a ses avantages et inconvénients qu'on étudiera après avoir détaillé leur mise en place.

2.4.2 Choix du répertoire d'archivage

- À faire quelle que soit la méthode d'archivage
- Attention aux droits d'écriture dans le répertoire
  - la commande configurée pour la copie doit pouvoir écrire dedans
  - et potentiellement y lire

Dans le cas de l'archivage historique, le serveur PostgreSQL va exécuter une commande qui va copier les journaux à l'extérieur de son répertoire de travail :

- sur un disque différent du même serveur ;
- sur un disque d'un autre serveur ;
- sur des bandes, un CDROM, etc.

Dans le cas de l'archivage avec `pg_receivewal`, c'est cet outil qui va écrire les journaux dans un répertoire de travail. Cette écriture ne peut se faire qu'en local. Cependant, le répertoire peut se trouver dans un montage NFS.

L'exemple pris ici utilise le répertoire `/mnt/nfs1/archivage` comme répertoire de copie. Ce répertoire est en fait un montage NFS. Il faut donc commencer par créer ce répertoire et s'assurer que l'utilisateur Unix (ou Windows) `postgres` peut écrire dedans :

```
# mkdir /mnt/nfs1/archivage
# chown postgres:postgres /mnt/nfs1/archivage
```

2.4.3 Processus archiver : configuration

- configuration (`postgresql.conf`)
  - `wal_level = replica`
  - `archive_mode = on` (ou `always`)
  - `archive_command = '… une commande …'`
  - ou : `archive_library = '… une bibliothèque …'` (v15+)
  - `archive_timeout = '… min'`
- Ne pas oublier de forcer l'écriture de l'archive sur disque
- Code retour de l'archivage entre 0 (ok) et 125

Après avoir créé le répertoire d'archivage, il faut configurer PostgreSQL pour lui indiquer comment archiver.

Niveau d'archivage :

La valeur par défaut de `wal_level` est adéquate :

```
wal_level = replica
```

Ce paramètre indique le niveau des informations écrites dans les journaux de transactions. Avec un niveau `minimal`, les journaux ne servent qu'à garantir la cohérence des fichiers de données en cas de crash. Dans le cas d'un archivage, il faut écrire plus d'informations, d'où la nécessité du niveau `replica` (qui est celui par défaut). Le niveau `logical`, nécessaire à la réplication logique[3], convient également.

Mode d'archivage :

Il s'active ainsi sur une instance seule ou primaire :

```
archive_mode = on
```

(La valeur `always` permet d'archiver depuis un secondaire). Le changement nécessite un redémarrage !

Commande d'archivage :

Enfin, une commande d'archivage doit être définie par le paramètre `archive_command`. `archive_command` sert à archiver un seul fichier à chaque appel.

PostgreSQL l'appelle une fois pour chaque fichier WAL, impérativement dans l'ordre des fichiers. En cas d'échec, elle est répétée indéfiniment jusqu'à réussite, avant de passer à l'archivage du fichier suivant. C'est la technique encore la plus utilisée.

(Noter qu'à partir de la version 15, il existe une alternative, avec l'utilisation du paramètre `archive_library`. Il est possible d'indiquer une bibliothèque partagée qui fera ce travail d'archivage. Une telle bibliothèque, écrite en C, devrait être plus puissante et performante. Un module basique est fourni avec PostgreSQL : basic_archive[4]. Notre blog présente un exemple fonctionnel de module d'archivage[5] utilisant une extension en C pour compresser les journaux de transactions. Mais en production, il vaudra mieux utiliser une bibliothèque fournie par un outil PITR reconnu. Cependant, à notre connaissance (en décembre 2023), aucun n'utilise encore cette fonctionnalité. L'utilisation simultanée de `archive_command` et `archive_library` est déconseillée, et interdite depuis PostgreSQL 16.)

Exemples d'archive_command :

PostgreSQL laisse le soin à l'administrateur de définir la méthode d'archivage des journaux de transactions suivant son contexte. Si vous utilisez un outil de sauvegarde, la commande vous sera probablement fournie. Une simple commande de copie suffit dans la plupart des cas. La directive `archive_command` peut alors être positionnée comme suit :

[3] https://dali.bo/w5_html
[4] https://docs.postgresql.fr/current/basic-archive.html
[5] https://blog.dalibo.com/2023/07/28/hackingpg2.html

```
archive_command = 'cp %p /mnt/nfs1/archivage/%f'
```

Le joker `%p` est remplacé par le chemin complet vers le journal de transactions à archiver, alors que le joker `%f` correspond au nom du journal de transactions une fois archivé.

En toute rigueur, une copie du fichier ne suffit pas. Par exemple, dans le cas de la commande `cp`, le nouveau fichier n'est pas immédiatement écrit sur disque. La copie est effectuée dans le cache disque du système d'exploitation. En cas de crash juste après la copie, il est tout à fait possible de perdre l'archive. Il est donc essentiel d'ajouter une étape de synchronisation du cache sur disque.

La commande d'archivage suivante est donnée dans la documentation officielle à titre d'exemple :

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p
↪  /mnt/server/archivedir/%f'
```

Cette commande a deux inconvénients. Elle ne garantit pas que les données seront synchronisées sur disque. De plus si le fichier existe ou a été copié partiellement à cause d'une erreur précédente, la copie ne s'effectuera pas.

Cette protection est une bonne chose. Cependant, il faut être vigilant lorsque l'on rétablit le fonctionnement de l'*archiver* suite à un incident ayant provoqué des écritures partielles dans le répertoire d'archive, comme une saturation de l'espace disque.

Il est aussi possible de placer dans `archive_command` le nom d'un script bash, perl ou autre. L'intérêt est de pouvoir faire plus qu'une simple copie. On peut y ajouter la demande de synchronisation du cache sur disque. Il peut aussi être intéressant de tracer l'action de l'archivage par exemple, ou encore de compresser le journal avant archivage.

Il faut s'assurer d'une seule chose : la commande d'archivage doit retourner 0 en cas de réussite et surtout une valeur différente de 0 en cas d'échec.
Si le code retour de la commande est compris entre 1 et 125, PostgreSQL va tenter périodiquement d'archiver le fichier jusqu'à ce que la commande réussisse (autrement dit, renvoie 0).
Tant qu'un fichier journal n'est pas considéré comme archivé avec succès, PostgreSQL ne le supprimera ou recyclera pas !
Il ne cherchera pas non plus à archiver les fichiers suivants.

De plus si le code retour de la commande est supérieur à 125, le processus `archiver` redémarrera, et l'erreur ne sera pas comptabilisée dans la vue `pg_stat_archiver` ! Ce cas de figure inclut les erreurs de type `command not found` associées aux codes retours 126 et 127, ou le cas de `rsync`, qui renvoie un code retour 255 en cas d'erreur de syntaxe ou de configuration du ssh.

Il est donc important de surveiller le processus d'archivage et de faire remonter les problèmes à un opérateur. Les causes d'échec sont nombreuses : problème réseau, montage inaccessible, erreur de paramétrage de l'outil, droits insuffisants ou expirés, génération de journaux trop rapide…

À titre d'exemple encore, les commandes fournies par pgBackRest ou barman ressemblent à ceci :

```
# pgBackRest
archive_command='/usr/bin/pgbackrest --stanza=prod archive-push %p'

# barman
archive_command='/usr/bin/barman-wal-archive backup prod %p'
```

Enfin, le paramétrage suivant archive « dans le vide ». Cette astuce est utilisée lors de certains dépannages, ou pour éviter le redémarrage que nécessiterait la désactivation de `archive_mode`.

```
archive_mode = on
archive_command = '/bin/true'
```

Période minimale entre deux archivages :

Si l'activité en écriture est très réduite, il peut se passer des heures entre deux archivages de journaux. Il est alors conseillé de forcer un archivage périodique, même si le journal n'a pas été rempli complètement, en indiquant un délai maximum entre deux archivages :

```
archive_timeout = '5min'
```

(La valeur par défaut, 0, désactive ce comportement.) Comme la taille d'un fichier journal, même incomplet, reste fixe (16 Mo par défaut), la consommation en terme d'espace disque sera plus importante (la compression par l'outil d'archivage peut compenser cela), et le temps de restauration plus long.

2.4.4 Processus archiver : lancement

- Redémarrage de PostgreSQL
  - si modification de `wal_level` et/ou `archive_mode`
- ou rechargement de la configuration

Il ne reste plus qu'à indiquer à PostgreSQL de recharger sa configuration pour que l'archivage soit en place (avec `SELECT pg_reload_conf();` ou la commande `reload` adaptée au système). Dans le cas où l'un des paramètres `wal_level` et `archive_mode` a été modifié, il faudra relancer PostgreSQL.

2.4.5 Processus archiver : supervision

- Vue `pg_stat_archiver`
- `pg_wal/archive_status/`
- Taille de `pg_wal`
  - si saturation : Arrêt !
- Traces

PostgreSQL archive les journaux impérativement dans l'ordre.

S'il y a un problème d'archivage d'un journal, les suivants ne seront pas archivés non plus, et vont s'accumuler dans `pg_wal` ! De plus, une saturation de la partition portant `pg_wal` mènera à l'arrêt de l'instance PostgreSQL !

La supervision se fait de quatre manières complémentaires.

Taille :

Si le répertoire `pg_wal` commence à grossir fortement, c'est que PostgreSQL n'arrive plus à recycler ses journaux de transactions : c'est un indicateur d'une commande d'archivage n'arrivant pas à faire son travail pour une raison ou une autre. Ce peut être temporaire si l'archivage est juste lent. Si l'archivage est bloqué, ce répertoire grossira indéfiniment.

Vue pg_stat_archiver :

La vue système `pg_stat_archiver` indique les derniers journaux archivés et les dernières erreurs. Dans l'exemple suivant, il y a eu un problème pendant quelques secondes, d'où 6 échecs, avant que l'archivage reprenne :

```
# SELECT * FROM pg_stat_archiver \gx

-[ RECORD 1 ]------+------------------------------
archived_count     | 156
last_archived_wal  | 000000020000000100000D9
last_archived_time | 2020-01-17 18:26:03.715946+00
failed_count       | 6
last_failed_wal    | 000000020000000100000D7
last_failed_time   | 2020-01-17 18:24:24.463038+00
stats_reset        | 2020-01-17 16:08:37.980214+00
```

Comme dit plus haut, pour que cette supervision soit fiable, la commande exécutée doit renvoyer un code retour inférieur ou égal à 125. Dans le cas contraire, le processus archiver redémarre et l'erreur n'apparaît pas dans la vue !

Traces :

On trouvera la sortie et surtout les messages d'erreurs du script d'archivage dans les traces (qui dépendent bien sûr du script utilisé) :

```
2020-01-17 18:24:18.427 UTC [15431] LOG:  archive command failed with exit code 3
2020-01-17 18:24:18.427 UTC [15431] DETAIL:  The failed archive command was:
  rsync pg_wal/000000020000000100000D7 /opt/pgsql/archives/000000020000000100000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:19.456 UTC [15431] LOG:  archive command failed with exit code 3
2020-01-17 18:24:19.456 UTC [15431] DETAIL:  The failed archive command was:
  rsync pg_wal/000000020000000100000D7 /opt/pgsql/archives/000000020000000100000D7
rsync: change_dir#3 "/opt/pgsql/archives" failed: No such file or directory (2)
rsync error: errors selecting input/output files, dirs (code 3) at main.c(695)
  [Receiver=3.1.2]
2020-01-17 18:24:20.463 UTC [15431] LOG:  archive command failed with exit code 3
```

C'est donc le premier endroit à regarder en cas de souci ou lors de la mise en place de l'archivage.

pg_wal/archive_status :

Enfin, on peut monitorer les fichiers présents dans `pg_wal/archive_status`. Les fichiers `.ready`, de taille nulle, indiquent en permanence quels sont les journaux prêts à être archivés. Théoriquement, leur nombre doit donc rester faible et retomber rapidement à 0 ou 1. Le service `ready_archives` de la sonde Nagios check_pgactivity[6] se base sur ce répertoire.

```
postgres=# SELECT * FROM pg_ls_dir ('pg_wal/archive_status') ORDER BY 1;

           pg_ls_dir
--------------------------------
 000000020000000100000DE.done
```

[6] https://github.com/OPMDG/check_pgactivity

```
000000020000000100000DF.done
000000020000000100000E0.done
000000020000000100000E1.ready
000000020000000100000E2.ready
000000020000000100000E3.ready
000000020000000100000E4.ready
000000020000000100000E5.ready
000000020000000100000E6.ready
00000002.history.done
```

2.4.6 pg_receivewal

- Archivage via le protocole de réplication
- Enregistre en local les journaux de transactions
- Permet de faire de l'archivage PITR
    - toujours au plus près du primaire
- Slots de réplication obligatoires

`pg_receivewal` est un outil permettant de se faire passer pour un serveur secondaire utilisant la réplication en flux (*streaming replication*) dans le but d'archiver en continu les journaux de transactions. Il fonctionne habituellement sur un autre serveur, où seront archivés les journaux. C'est une alternative à l'`archiver`.

Comme il utilise le protocole de réplication, les journaux archivés ont un retard bien inférieur à celui induit par la configuration du paramètre `archive_command` ou du paramètre `archive_library`, les journaux de transactions étant écrits au fil de l'eau avant d'être complets. Cela permet donc de faire de l'archivage PITR avec une perte de données minimum en cas d'incident sur le serveur primaire. On peut même utiliser une réplication synchrone (paramètres `synchronous_commit` et `synchronous_standby_names`) pour ne perdre aucune transaction, si l'on accepte un impact certain sur la latence des transactions.

Cet outil utilise les mêmes options de connexion que la plupart des outils PostgreSQL, avec en plus l'option `-D` pour spécifier le répertoire où sauvegarder les journaux de transactions. L'utilisateur spécifié doit bien évidemment avoir les attributs `LOGIN` et `REPLICATION`.

Comme il s'agit de conserver toutes les modifications effectuées par le serveur dans le cadre d'une sauvegarde permanente, il est nécessaire de s'assurer qu'on ne puisse pas perdre des journaux de transactions. Il n'y a qu'un seul moyen pour cela : utiliser la technologie des slots de réplication. En utilisant un slot de réplication, `pg_receivewal` s'assure que le serveur ne va pas recycler des journaux dont `pg_receivewal` n'aurait pas reçu les enregistrements. On retrouve donc le risque d'accumulation des journaux sur le serveur principal si `pg_receivewal` ne fonctionne pas.

Voici l'aide de cet outil en v15 :

```
$ pg_receivewal --help
pg_receivewal reçoit le flux des journaux de transactions PostgreSQL.

Usage :
  pg_receivewal [OPTION]...

Options :
  -D, --directory=RÉPERTOIRE     reçoit les journaux de transactions dans ce
                                 répertoire
  -E, --endpos=LSN               quitte après avoir reçu le LSN spécifié
      --if-not-exists            ne pas renvoyer une erreur si le slot existe
                                 déjà lors de sa création
  -n, --no-loop                  ne boucle pas en cas de perte de la connexion
      --no-sync                  n'attend pas que les modifications soient
                                 proprement écrites sur disque
  -s, --status-interval=SECS     durée entre l'envoi de paquets de statut au
                                 (par défaut 10)
  -S, --slot=NOMREP              slot de réplication à utiliser
      --synchronous              vide le journal de transactions immédiatement
                                 après son écriture
  -v, --verbose                  affiche des messages verbeux
  -V, --version                  affiche la version puis quitte
  -Z, --compress=METHOD[:DETAIL]
                                 compresse comme indiqué
  -?, --help                     affiche cette aide puis quitte

Options de connexion :
  -d, --dbname=CHAÎNE_CONNEX     chaîne de connexion
  -h, --host=HÔTE                hôte du serveur de bases de données ou
                                 répertoire des sockets
  -p, --port=PORT                numéro de port du serveur de bases de données
  -U, --username=UTILISATEUR     se connecte avec cet utilisateur
  -w, --no-password              ne demande jamais le mot de passe
  -W, --password                 force la demande du mot de passe (devrait
                                 survenir automatiquement)

Actions optionnelles :
      --create-slot              crée un nouveau slot de réplication
                                 (pour le nom du slot, voir --slot)
      --drop-slot                supprime un nouveau slot de réplication
                                 (pour le nom du slot, voir --slot)

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>.
Page d'accueil de PostgreSQL : <https://www.postgresql.org/>
```

2.4.7 pg_receivewal - configuration serveur

- `postgresql.conf` :

```
# configuration  par défaut
max_wal_senders = 10
max_replication_slots = 10
```

- `pg_hba.conf` :

```
host  replication  repli_user  192.168.0.0/24  scram-sha-256
```

- Utilisateur de réplication :

```
CREATE ROLE repli_user LOGIN REPLICATION PASSWORD 'supersecret'
```

Le paramètre `max_wal_senders` indique le nombre maximum de connexions de réplication sur le serveur. Logiquement, une valeur de 1 serait suffisante, mais il faut compter sur quelques soucis réseau qui pourraient faire perdre la connexion à `pg_receivewal` sans que le serveur primaire n'en soit mis au courant, et du fait que certains autres outils peuvent utiliser la réplication. `max_replication_slots` indique le nombre maximum de slots de réplication. Pour ces deux paramètres, le défaut est 10 et suffit dans la plupart des cas.

Si l'on modifie un de ces paramètres, il est nécessaire de redémarrer le serveur PostgreSQL.

Les connexions de réplication nécessitent une configuration particulière au niveau des accès. D'où la modification du fichier `pg_hba.conf`. Le sous-réseau (192.168.0.0/24) est à modifier suivant l'adressage utilisé. Il est d'ailleurs préférable de n'indiquer que le serveur où est installé `pg_receivewal` (plutôt que l'intégralité d'un sous-réseau).

L'utilisation d'un utilisateur de réplication n'est pas obligatoire mais fortement conseillée pour des raisons de sécurité.

2.4.8 pg_receivewal - redémarrage du serveur

- Redémarrage de PostgreSQL
- Slot de réplication

```
SELECT pg_create_physical_replication_slot('archivage');
```

Enfin, nous devons créer le slot de réplication qui sera utilisé par `pg_receivewal`. La fonction `pg_create_physical_replication_slot()` est là pour ça. Il est à noter que la liste des slots est disponible dans le catalogue système `pg_replication_slots`.

2.4.9 pg_receivewal - lancement de l'outil

- Exemple de lancement

```
pg_receivewal -D /data/archives -S archivage
```

- Journaux créés en temps réel dans le répertoire de stockage
- Mise en place d'un script de démarrage
- S'il n'arrive pas à joindre le serveur primaire
  - Au démarrage de l'outil : `pg_receivewal` s'arrête
  - En cours d'exécution : `pg_receivewal` tente de se reconnecter
- Nombreuses options

On peut alors lancer `pg_receivewal` :

```
pg_receivewal -h 192.168.0.1 -U repli_user -D /data/archives -S archivage
```

Les journaux de transactions sont alors créés en temps réel dans le répertoire indiqué (ici, `/data/archives`) :

```
-rwx------  1 postgres postgres  16MB juil. 27 00000001000000000000000E*
-rwx------  1 postgres postgres  16MB juil. 27 00000001000000000000000F*
-rwx------  1 postgres postgres  16MB juil. 27 00000001000000000000000.partial*
```

En cas d'incident sur le serveur primaire, il est alors possible de partir d'une sauvegarde physique et de rejouer les journaux de transactions disponibles (sans oublier de supprimer l'extension `.partial` du dernier journal).

Il faut mettre en place un script de démarrage pour que `pg_receivewal` soit redémarré en cas de redémarrage du serveur.

2.4.10 Avantages et inconvénients

- Méthode archiver
  - simple à mettre en place
  - perte au maximum d'un journal de transactions
- Méthode `pg_receivewal`
  - mise en place plus complexe
  - perte minimale voire nulle

La méthode archiver est la méthode la plus simple à mettre en place. Elle se lance au lancement du serveur PostgreSQL, donc il n'est pas nécessaire de créer et installer un script de démarrage. Cependant, un journal de transactions n'est archivé que quand PostgreSQL l'ordonne, soit parce qu'il a rempli le journal en question, soit parce qu'un utilisateur a forcé un changement de journal (avec la fonction `pg_switch_wal` ou suite à un `pg_backup_stop`), soit parce que le délai maximum entre deux archivages a été dépassé (paramètre `archive_timeout`). Il est donc possible de perdre un grand nombre de transactions (même si cela reste bien inférieur à la perte qu'une restauration d'une sauvegarde logique occasionnerait).

La méthode `pg_receivewal` est plus complexe à mettre en place. Il faut exécuter ce démon, généralement sur un autre serveur. Un script de démarrage doit donc être configuré. Par contre, elle a le gros avantage de ne perdre pratiquement aucune transaction, surtout en mode synchrone. Les enregistrements de transactions sont envoyés en temps réel à `pg_receivewal`. Ce dernier les place dans un fichier de suffixe `.partial`, qui est ensuite renommé pour devenir un journal de transactions complet.

2.5 SAUVEGARDE PITR MANUELLE

- 3 étapes :
  - fonction de démarrage
  - copie des fichiers par outil externe
  - fonction d'arrêt
- Exclusive : simple… & obsolète ! (< v15)
- Concurrente : plus complexe à scripter
- Aucun impact pour les utilisateurs ; pas de verrou
- Préférer des outils dédiés qu'un script maison

Une fois l'archivage en place, une sauvegarde à chaud a lieu en trois temps :

- l'appel à la fonction de démarrage ;
- la copie elle-même par divers outils externes (PostgreSQL ne s'en occupe pas) ;
- l'appel à la fonction d'arrêt.

La fonction de démarrage s'appelle `pg_backup_start()` à partir de la version 15 mais avait pour nom `pg_start_backup()` auparavant. De la même façon, la fonction d'arrêt s'appelle `pg_backup_stop()` à partir de la version 15, mais `pg_stop_backup()` avant.

La sauvegarde exclusive était la plus simple, et cela en faisait le choix par défaut. Il suffisait d'appeler les fonctions concernées avant et après la copie des fichiers. Il ne pouvait y en avoir qu'une à la fois. Elle ne fonctionnait que depuis un primaire.

À cause de ces limites et de différents problèmes, , la sauvegarde exclusive est déclarée obsolète depuis la 9.6, et n'est plus disponible depuis la version 15. Même sur les versions antérieures, il est conseillé d'utiliser dès maintenant des scripts utilisant les sauvegardes concurrentes.

Tout ce qui suit ne concerne plus que la sauvegarde concurrente.

La sauvegarde concurrente, apparue avec PostgreSQL 9.6, peut être lancée plusieurs fois en parallèle. C'est utile pour créer des secondaires alors qu'une sauvegarde physique tourne, par exemple. Elle est nettement plus complexe à gérer par script. Elle peut être exécutée depuis un serveur secondaire, ce qui allège la charge sur le primaire.

Pendant la sauvegarde, l'utilisateur ne verra aucune différence (à part peut-être la conséquence d'I/O saturées pendant la copie). Aucun verrou n'est posé. Lectures, écritures, suppression et création de tables, archivage de journaux et autres opérations continuent comme si de rien n'était.

La description du mécanisme qui suit est essentiellement destinée à la compréhension et l'expérimentation. En production, un script maison reste une possibilité, mais préférez des outils dédiés et fiables : pg_basebackup, pgBackRest…

Les sauvegardes manuelles servent cependant encore quand on veut utiliser une sauvegarde par snapshot, ou avec `rsync` (car pg_basebackup ne sait pas synchroniser vers une sauvegarde interrompue ou ancienne), et quand les outils conseillés ne sont pas utilisables ou disponibles sur le système.

2.5.1 Sauvegarde manuelle - 1/3 : pg_backup_start

`SELECT pg_backup_start (`

- `un_label` : texte
- `fast` : forcer un checkpoint ?

`)`

L'exécution de `pg_backup_start()` peut se faire depuis n'importe quelle base de données de l'instance.

(Rappelons que pour les versions avant la 15, la fonction s'appelle `pg_start_backup()`. Pour effectuer une sauvegarde non-exclusive avec ces versions, il faudra positionner un troisième paramètre[7] à `false`.)

Le label (le texte en premier argument) n'a aucune importance pour PostgreSQL (il ne sert qu'à l'administrateur, pour reconnaître le backup).

Le deuxième argument est un booléen qui permet de demander un *checkpoint* immédiat, si l'on est pressé et si un pic d'I/O n'est pas gênant. Sinon il faudra attendre souvent plusieurs minutes (selon la configuration du déclenchement du prochain checkpoint, dépendant des paramètres `checkpoint_timeout` et `max_wal_size` et de la rapidité d'écriture imposée par `checkpoint_completion_target`).

La session qui exécute la commande `pg_backup_start()` doit être la même que celle qui exécutera plus tard `pg_backup_stop()`. Nous verrons que cette dernière fonction fournira de quoi créer deux fichiers, qui devront être nommés `backup_label` et `tablespace_map`. Si la connexion est interrompue avant `pg_backup_stop()`, alors la sauvegarde doit être considérée comme invalide.

[7] https://docs.postgresql.fr/14/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP-EXCLUSIVE

En plus de rester connectés à la base, les scripts qui gèrent la sauvegarde concurrente doivent donc récupérer et conserver les informations renvoyées par la commande de fin de sauvegarde.

La sauvegarde PITR est donc devenue plus complexe au fil des versions, et il est donc recommandé d'utiliser plutôt pg_basebackup ou des outils la supportant (barman, pgBackRest…).

2.5.2 Sauvegarde manuelle - 2/3 : copie des fichiers

- Cas courant : snapshot
  - cohérence ? redondance ?
- Sauvegarde des fichiers **à chaud**
  - répertoire principal des données
  - tablespaces
- Copie forcément incohérente (la restauration des journaux corrigera)
- `rsync` et autres outils
- Ignorer :
  - `postmaster.pid`, `log`, `pg_wal`, `pg_replslot` et quelques autres
- Ne pas oublier : configuration !

La deuxième étape correspond à la sauvegarde des fichiers. Le choix de l'outil dépend de l'administrateur. Cela n'a aucune incidence au niveau de PostgreSQL.

La sauvegarde doit comprendre aussi les tablespaces si l'instance en dispose.

Snapshot :

Il est possible d'effectuer cette étape de copie des fichiers par snapshot au niveau de la baie, de l'hyperviseur, ou encore de l'OS (LVM, ZFS…).

Un snaphost cohérent, y compris entre les tablespaces, permet théoriquement de réaliser une sauvegarde en se passant des étapes `pg_backup_start()` et `pg_backup_stop()`. La restauration de ce snapshot équivaudra pour PostgreSQL à un redémarrage brutal.
Pour une sauvegarde PITR, il faudra cependant toujours encadrer le snapshot des appels aux fonctions de démarrage et d'arrêt ci-dessus, et c'est généralement ce que font les outils comme Veeam ou Tina. L'utilisation d'un tel outil implique de vérifier qu'il sait gérer les sauvegardes non exclusives pour utiliser PostgreSQL 15 et supérieurs.

Le point noir de la sauvegarde par snapshot est d'être liée au même système matériel que l'instance PostgreSQL (disque, hyperviseur, datacenter…) Une défaillance grave du matériel peut donc emporter, corrompre ou bloquer la sauvegarde en même temps que les données originales. La sécurité de l'instance est donc reportée sur celle de l'infrastructure sous-jacente. Une copie parallèle des données de manière plus classique est conseillée pour éviter un désastre total.

Copie manuelle :

La sauvegarde se fait à chaud : il est donc possible que pendant ce temps des fichiers changent, disparaissent avant d'être copiés ou apparaissent sans être copiés. Cela n'a pas d'importance en soi car les journaux de transactions corrigeront cela (leur archivage doit donc commencer **avant** le début de la sauvegarde et se poursuivre sans interruption jusqu'à la fin).
Il **faut** s'assurer que l'outil de sauvegarde supporte cela, c'est-à-dire qu'il soit capable de différencier les codes d'erreurs dus à « des fichiers ont bougé ou disparu lors de la sauvegarde » des autres erreurs techniques. `tar` par exemple convient : il retourne 1 pour le premier cas d'erreur, et 2 quand l'erreur est critique. `rsync` est très courant également.

Sur les plateformes Microsoft Windows, peu d'outils sont capables de copier des fichiers en cours de modification. Assurez-vous d'en utiliser un possédant cette fonctionnalité (il existe différents émulateurs des outils GNU sous Windows). Le plus sûr et simple est sans doute de renoncer à une copie manuelle des fichiers et d'utiliser `pg_basebackup`.

Exclusions :

Des fichiers et répertoires sont à ignorer, pour gagner du temps ou faciliter la restauration. Voici la liste exhaustive (disponible aussi dans la documentation officielle[8]) :

- `postmaster.pid`, `postmaster.opts`, `pg_internal.init` ;
- les fichiers de données des tables non journalisées (*unlogged*) ;
- `pg_wal`, ainsi que les sous-répertoires (mais à archiver séparément !) ;
- `pg_replslot` : les slots de réplication seront au mieux périmés, au pire gênants sur l'instance restaurée ;
- `pg_dynshmem`, `pg_notify`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` et `pg_subtrans` ne doivent pas être copiés (ils contiennent des informations propres à l'instance, ou qui ne survivent pas à un redémarrage) ;
- les fichiers et répertoires commençant par `pgsql_tmp` (fichiers temporaires) ;
- les fichiers autres que les fichiers et les répertoires standards (donc pas les liens symboliques).

On n'oubliera pas les fichiers de configuration s'ils ne sont pas dans le PGDATA.

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

2.5.3 Sauvegarde manuelle - 3/3 : pg_backup_stop

Ne pas oublier !!
`SELECT * FROM pg_backup_stop (`

- `true` : attente de l'archivage

`)`

La dernière étape correspond à l'exécution de la procédure stockée `SELECT * FROM pg_backup_stop()`.

N'oubliez pas d'exécuter `pg_backup_stop()`, de vérifier qu'il finit avec succès et de récupérer les informations qu'il renvoie !
Cet oubli trop courant rend vos sauvegardes inutilisables !

PostgreSQL va alors :

- marquer cette fin de backup dans le journal des transactions (étape capitale pour la restauration) ;
- forcer la finalisation du journal de transactions courant et donc son archivage, afin que la sauvegarde (fichiers + archives) soit utilisable même en cas de crash juste l'appel à la fonction : `pg_backup_stop()` ne rendra pas la main (par défaut) tant que ce dernier journal n'aura pas été archivé avec succès.

La fonction renvoie :

- le *lsn* de fin de backup ;
- un champ destiné au fichier `backup_label` ;
- un champ destiné au fichier `tablespace_map`.

```
# SELECT * FROM pg_stop_backup() \gx

NOTICE:  all required WAL segments have been archived
-[ RECORD 1 ]----------------------------------------------------------
lsn        | 22/2FE5C788
labelfile  | START WAL LOCATION: 22/2B000028 (file 000000010000002200000002B)+
           | CHECKPOINT LOCATION: 22/2B000060                             +
           | BACKUP METHOD: streamed                                      +
           | BACKUP FROM: master                                          +
           | START TIME: 2019-12-16 13:53:41 CET                         +
           | LABEL: rr                                                   +
           | START TIMELINE: 1                                           +
           |
```

```
spcmapfile | 134141 /tbl/froid                                              +
           | 134152 /tbl/quota                                               +
           |
```

Ces informations se retrouvent aussi dans un fichier `.backup` mêlé aux journaux :

```
# cat /var/lib/postgresql/12/main/pg_wal/000000010000000220000002B.00000028.backup

START WAL LOCATION: 22/2B000028 (file 000000010000000220000002B)
STOP WAL LOCATION: 22/2FE5C788 (file 000000010000000220000002F)
CHECKPOINT LOCATION: 22/2B000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2019-12-16 13:53:41 CET
LABEL: rr
START TIMELINE: 1
STOP TIME: 2019-12-16 13:54:04 CET
STOP TIMELINE: 1
```

Il faudra créer le fichier `tablespace_map` avec le contenu du champ `spcmapfile` :

```
134141 /tbl/froid
134152 /tbl/quota
```

… ce qui n'est pas trivial à scripter.

Ces deux fichiers devront être placés dans la sauvegarde, pour être présent d'entrée dans le PGDATA du serveur restauré.

À partir du moment où `pg_backup_stop()` rend la main, il est possible de restaurer la sauvegarde obtenue puis de rejouer les journaux de transactions suivants en cas de besoin, sur un autre serveur ou sur ce même serveur.

Tous les journaux archivés avant celui précisé par le champ `START WAL LOCATION` dans le fichier `backup_label` ne sont plus nécessaires pour la récupération de la sauvegarde du système de fichiers et peuvent donc être supprimés. Attention, il y a plusieurs compteurs hexadécimaux différents dans le nom du fichier journal, qui ne sont pas incrémentés de gauche à droite.

2.5.4 Sauvegarde de base à chaud : pg_basebackup

Outil de sauvegarde pouvant aussi servir au sauvegarde basique

- Backup de base ici **sans** les journaux :

```
$ pg_basebackup --format=tar --wal-method=none \
 --checkpoint=fast --progress -h 127.0.0.1 -U sauve \
 -D /var/lib/postgresql/backups/
```

`pg_basebackup` a été décrit plus haut. Il a l'avantage d'être simple à utiliser, de savoir quels fichiers ne pas copier, de fiabiliser la sauvegarde par un slot de réplication. Il ne réclame en général pas de configuration supplémentaire.

Si l'archivage est déjà en place, copier les journaux est inutile (`--wal-method=none`). Nous verrons plus tard comment lui indiquer où les chercher.

L'inconvénient principal de `pg_basebackup` reste son incapacité à reprendre une sauvegarde interrompue ou à opérer une sauvegarde différentielle ou incrémentale.

2.5.5 Fréquence de la sauvegarde de base

- Dépend des besoins
- De tous les jours à tous les mois
- Plus elles sont espacées, plus la restauration est longue
  - et plus le risque d'un journal corrompu ou absent est important

La fréquence dépend des besoins. Une sauvegarde par jour est le plus commun, mais il est possible d'espacer encore plus la fréquence.

Cependant, il faut conserver à l'esprit que plus la sauvegarde est ancienne, plus la restauration sera longue car un plus grand nombre de journaux seront à rejouer.

2.5.6 Suivi de la sauvegarde de base

- Vue `pg_stat_progress_basebackup`
  - à partir de la v13

La version 13 permet de suivre la progression de la sauvegarde de base, quelque soit l'outil utilisé à condition qu'il passe par le protocole de réplication.

Cela permet ainsi de savoir à quelle phase la sauvegarde se trouve, quelle volumétrie a été envoyée, celle à envoyer, etc.

2.6 RESTAURER UNE SAUVEGARDE PITR

- Une procédure relativement simple
- Mais qui doit être effectuée rigoureusement

La restauration se déroule en trois voire quatre étapes suivant qu'elle est effectuée sur le même serveur ou sur un autre serveur.

2.6.1 Restaurer une sauvegarde PITR (1/5)

- S'il s'agit du même serveur
  - arrêter PostgreSQL
  - supprimer le répertoire des données
  - supprimer les tablespaces

Dans le cas où la restauration a lieu sur le même serveur, quelques étapes préliminaires sont à effectuer.

Il faut arrêter PostgreSQL s'il n'est pas arrêté. Cela arrivera quand la restauration a pour but, par exemple, de récupérer des données qui ont été supprimées par erreur.

Ensuite, il faut supprimer (ou archiver) l'ancien répertoire des données pour pouvoir y placer la sauvegarde des fichiers. Écraser l'ancien répertoire n'est pas suffisant, il faut le supprimer, ainsi que les répertoires des tablespaces au cas où l'instance en possède.

2.6.2 Restaurer une sauvegarde PITR (2/5)

- Restaurer les fichiers de la sauvegarde
- Supprimer les fichiers compris dans le répertoire `pg_wal` restauré
  - ou mieux, ne pas les avoir inclus dans la sauvegarde initialement
- Restaurer le dernier journal de transactions connu (si disponible)

La sauvegarde des fichiers peut enfin être restaurée. Il faut bien porter attention à ce que les fichiers soient restaurés au même emplacement, tablespaces compris.

Une fois cette étape effectuée, il peut être intéressant de faire un peu de ménage. Par exemple, le fichier `postmaster.pid` peut poser un problème au démarrage. Conserver les journaux applicatifs n'est pas en soi un problème mais peut porter à confusion. Il est donc préférable de les supprimer. Quant aux journaux de transactions compris dans la sauvegarde, bien que ceux en provenance des archives seront utilisés même s'ils sont présents aux deux emplacements, il est préférable de les supprimer. La commande sera similaire à celle-ci :

```
$ rm postmaster.pid log/* pg_wal/[0-9A-F]*
```

Enfin, s'il est possible d'accéder au journal de transactions courant au moment de l'arrêt de l'ancienne instance, il est intéressant de le restaurer dans le répertoire `pg_wal` fraîchement nettoyé. Ce dernier sera pris en compte en toute fin de restauration des journaux depuis les archives et permettra donc de restaurer les toutes dernières transactions validées sur l'ancienne instance, mais pas encore archivées.

2.6.3 Restaurer une sauvegarde PITR (3/5)

- Indiquer qu'on est en restauration
- Commande de restauration
  - `restore_command = '… une commande …'`
  - dans `postgresql.[auto.]conf`

Quand PostgreSQL démarre après avoir subi un arrêt brutal, il ne restaure que les journaux en place dans `pg_wal`, puis il s'ouvre en écriture. Pour une restauration, il faut lui indiquer qu'il doit aller demander les journaux quelque part, et les rejouer tous jusqu'à épuisement, avant de s'ouvrir.

Pour cela, il suffit de créer un fichier vide `recovery.signal` dans le répertoire des données.

Pour la récupération des journaux, le paramètre essentiel est `restore_command`. Il contient une commande symétrique des paramètres `archive_command` (ou `archive_library`) pour l'archivage. Il s'agit d'une commande copiant un journal dans le `pg_wal`. Cette commande est souvent fournie par l'outil de sauvegarde PITR s'il y en a un. Si nous poursuivons notre exemple, ce paramètre pourrait être :

```
restore_command = 'cp /mnt/nfs1/archivage/%f %p'
```

Cette commande sera appelée après la restauration de chaque journal pour récupérer le suivant, qui sera restauré et ainsi de suite. Il n'y a aucune paralléllisation prévue, mais des outils de sauvegarde PITR peuvent le faire au sein de la commande.

Si le but est de restaurer tous les journaux archivés, il n'est pas nécessaire d'aller plus loin dans la configuration. La restauration se poursuivra jusqu'à ce que `restore_command` tombe en erreur, ce qui signifie l'épuisement de tous les journaux disponibles, et la fin de la restauration.

Au cas où vous rencontreriez une instance en version 11 ou antérieure, il faut savoir que la restauration se paramétrait dans un fichier texte dans le PGDATA, contenant `recovery_command` et éventuellement les options de restauration.

2.6.4 Restaurer une sauvegarde PITR (4/5)

- Jusqu'où restaurer :
  - `recovery_target_name`, `recovery_target_time`
  - `recovery_target_xid`, `recovery_target_lsn`
  - `recovery_target_inclusive`
- Le backup de base doit être antérieur !
- Suivi de timeline :
  - `recovery_target_timeline` : `latest` ?
- Et on fait quoi ?
  - `recovery_target_action` : `pause`
  - `pg_wal_replay_resume` pour ouvrir immédiatement
  - ou modifier & redémarrer

Si l'on ne veut pas simplement restaurer tous les journaux, par exemple pour s'arrêter avant une fausse manipulation désastreuse, plusieurs paramètres permettent de préciser le point d'arrêt :

- jusqu'à un certain nom, grâce au paramètre `recovery_target_name` (le nom correspond à un label enregistré précédemment dans les journaux de transactions grâce à la fonction `pg_create_restore_point()`) ;
- jusqu'à une certaine heure, grâce au paramètre `recovery_target_time` ;
- jusqu'à un certain identifiant de transaction, grâce au paramètre `recovery_target_xid`, numéro de transaction qu'il est possible de chercher dans les journaux eux-mêmes grâce à l'utilitaire `pg_waldump` ;
- jusqu'à un certain LSN (*Log Sequence Number*[9]), grâce au paramètre `recovery_target_lsn`,

[9] https://docs.postgresql.fr/current/datatype-pg-lsn.html

que là aussi on doit aller chercher dans les journaux eux-mêmes.

Avec le paramètre `recovery_target_inclusive` (par défaut à `true`), il est possible de préciser si la restauration se fait en incluant les transactions au nom, à l'heure ou à l'identifiant de transaction demandé, ou en les excluant.

Dans les cas complexes, nous verrons plus tard que choisir la *timeline* peut être utile (avec `recovery_target_timeline`, en général à `latest`).

Ces restaurations ponctuelles ne sont possibles que si elles correspondent à un état cohérent d'**après** la fin du *base backup*, soit après le moment du `pg_stop_backup`.
Si l'on a un historique de plusieurs sauvegardes, il faudra en choisir une antérieure au point de restauration voulu. Ce n'est pas forcément la dernière. Les outils ne sont pas forcément capables de deviner la bonne sauvegarde à restaurer.

Il est possible de demander à la restauration de s'arrêter une fois arrivée au stade voulu avec :

```
recovery_target_action = pause
```

C'est même l'action par défaut si une des options d'arrêt ci-dessus a été choisie : cela permet à l'utilisateur de vérifier que le serveur est bien arrivé au point qu'il désirait. Les alternatives sont `promote` et `shutdown`.

Si la cible est atteinte mais que l'on décide de continuer la restauration jusqu'à un autre point (évidemment postérieur), il faut modifier la cible de restauration dans le fichier de configuration, et **redémarrer** PostgreSQL. C'est le seul moyen de rejouer d'autres journaux sans ouvrir l'instance en écriture.

Si l'on est arrivé au point de restauration voulu, un message de ce genre apparaît :

```
LOG:  recovery stopping before commit of transaction 8693270, time 2021-09-02
↪  11:46:35.394345+02
LOG:  pausing at the end of recovery
HINT:  Execute pg_wal_replay_resume() to promote.
```

(Le terme *promote* pour une restauration est un peu abusif.) `pg_wal_replay_resume()` — malgré ce que pourrait laisser croire son nom ! — provoque ici l'arrêt immédiat de la restauration, donc ignore les opérations contenues dans les WALs que l'on n'a pas souhaités restaurer, puis le serveur s'ouvre en écriture sur une nouvelle timeline.

Attention : jusque PostgreSQL 12 inclus, si un `recovery_target` était spécifié mais n'est toujours *pas* atteint à la fin du rejeu des archives, alors le mode *recovery* se terminait et le serveur est promu sans erreur, et ce, même si `recovery_target_action` a la valeur `pause` ! (À condition, bien sûr, que le point de cohérence ait tout de même été dépassé.) Il faut donc être vigilant quant aux messages dans le fichier de trace de PostgreSQL !
À partir de PostgreSQL 13, l'instance détecte le problème et s'arrête avec un message `FATAL` : la restauration ne s'est pas déroulée comme attendu. S'il manque juste certains journaux de transactions, cela permet de relancer PostgreSQL après correction de l'oubli.

La documentation officielle complète sur le sujet est sur le site du projet[10].

2.6.5 Restaurer une sauvegarde PITR (5/5)

- Démarrer PostgreSQL
- Rejeu des journaux
- Vérifier que le point de cohérence est atteint !

La dernière étape est particulièrement simple. Il suffit de démarrer PostgreSQL. PostgreSQL va comprendre qu'il doit rejouer les journaux de transactions.

Les journaux doivent se dérouler au moins jusqu'à rencontrer le « point de cohérence », c'est-à-dire la mention insérée par `pg_backup_stop()`. Avant cela, il n'est pas possible de savoir si les fichiers issus du *base backup* sont à jour ou pas, et il est impossible de démarrer l'instance avant ce point. Le message apparaît dans les traces et, dans le doute, on doit vérifier sa présence :

```
2020-01-17 16:08:37.285 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000031"…
2020-01-17 16:08:37.789 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000032"…
2020-01-17 16:08:37.949 UTC [15221] LOG: consistent recovery state reached
                                         at 1/32BFDD88
2020-01-17 16:08:37.949 UTC [15217] LOG: database system is ready to accept
                                         read only connections
2020-01-17 16:08:38.009 UTC [15221] LOG: restored log file
 ↪  "000000010000000100000033"…
```

PostgreSQL continue ensuite jusqu'à arriver à la limite fixée, jusqu'à ce qu'il ne trouve plus de journal à rejouer (`restore_command` tombe en erreur), ou que le bloc de journal lu soit incohérent (ce qui

[10] https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET

indique qu'on est arrivé à la fin d'un journal qui n'a pas été terminé, le journal courant au moment du crash par exemple). PostgreSQL vérifie qu'il n'existe pas une *timeline* supérieure sur laquelle basculer (par exemple s'il s'agit de la deuxième restauration depuis la sauvegarde du PGDATA).

Puis il va s'ouvrir en écriture (sauf si vous avez demandé `recovery_target_action = pause`).

```
2020-01-17 16:08:45.938 UTC [15221] LOG: restored log file "0000000100000000000000003C"
                                         from archive
2020-01-17 16:08:46.116 UTC [15221] LOG: restored log file
 ↪  "0000000100000000000000003D"…
2020-01-17 16:08:46.547 UTC [15221] LOG: restored log file
 ↪  "0000000100000000000000003E"…
2020-01-17 16:08:47.262 UTC [15221] LOG: restored log file
 ↪  "0000000100000000000000003F"…
2020-01-17 16:08:47.842 UTC [15221] LOG: invalid record length at 1/3F0000A0:
                                         wanted 24, got 0
2020-01-17 16:08:47.842 UTC [15221] LOG: redo done at 1/3F000028
2020-01-17 16:08:47.842 UTC [15221] LOG: last completed transaction was
                                         at log time 2020-01-17 14:59:30.093491+00
2020-01-17 16:08:47.860 UTC [15221] LOG: restored log file
 ↪  "0000000100000000000000003F"…
cp: cannot stat '/opt/pgsql/archives/00000002.history': No such file or directory
2020-01-17 16:08:47.966 UTC [15221] LOG:  selected new timeline ID: 2
2020-01-17 16:08:48.179 UTC [15221] LOG:  archive recovery complete
cp: cannot stat '/opt/pgsql/archives/00000001.history': No such file or directory
2020-01-17 16:08:51.613 UTC [15217] LOG:  database system is ready
                                          to accept connections
```

Le fichier `recovery.signal` est effacé pour ne pas poser problème en cas de crash immédiat. (Ne l'effacez jamais manuellement !)

Le fichier `backup_label` d'une sauvegarde exclusive est renommé en `backup_label.old`.

2.6.6 Restauration PITR : durée

- Durée dépendante du nombre de journaux
  - rejeu séquentiel des WAL
- Accéléré en version 15 (*prefetch*)

La durée de la restauration est fortement dépendante du nombre de journaux. Ils sont rejoués séquentiellement. Mais avant cela, un fichier journal peut devoir être récupéré, décompressé, et restauré dans `pg_wal`.

Il est donc préférable qu'il n'y ait pas trop de journaux à rejouer, et donc qu'il n'y ait pas trop d'espaces entre sauvegardes complètes successives.

La version 15 a optimisé le rejeu en permettant l'activation du *prefetch* des blocs de données lors du rejeu des journaux.

Un outil comme pgBackRest en mode asynchrone permet de paralléliser la récupération des journaux, ce qui permet de les récupérer via le réseau et de les décompresser par avance pendant que PostgreSQL traite les journaux précédents.

2.6.7 Restauration PITR : différentes timelines

- Fin de *recovery* => changement de *timeline* :
  - l'historique des données prend une autre voie
  - le nom des WAL change
  - fichier `.history`
- Permet plusieurs restaurations PITR à partir du même *basebackup*
- Choix : `recovery_target_timeline`
  - défaut : `latest`

Lorsque le mode *recovery* s'arrête, au point dans le temps demandé ou faute d'archives disponibles, l'instance accepte les écritures. De nouvelles transactions se produisent alors sur les différentes bases de données de l'instance. Dans ce cas, l'historique des données prend un chemin différent par rapport aux archives de journaux de transactions produites avant la restauration. Par exemple, dans ce nouvel historique, il n'y a pas le `DROP TABLE` malencontreux qui a imposé de restaurer les données. Cependant, cette transaction existe bien dans les archives des journaux de transactions.

On a alors plusieurs historiques des transactions, avec des « bifurcations » aux moments où on a réalisé des restaurations. PostgreSQL permet de garder ces historiques grâce à la notion de *timeline*. Une *timeline* est donc l'un de ces historiques, elle se matérialise par un ensemble de journaux de transactions, identifiée par un numéro. Le numéro de la *timeline* est le premier nombre hexadécimal du nom des segments de journaux de transactions (le second est le numéro du journal et le troisième le numéro du segment). Lorsqu'une instance termine une restauration PITR, elle peut archiver immédiatement ces journaux de transactions au même endroit, les fichiers ne seront pas écrasés vu qu'ils seront nommés différemment. Par exemple, après une restauration PITR s'arrêtant à un point situé dans le segment `000000010000000000000009` :

```
$ ls -1 /backup/postgresql/archived_wal/
000000010000000010000003C
000000010000000010000003D
000000010000000010000003E
000000010000000010000003F
000000010000000100000040
00000002.history
```

```
000000020000000010000003F
000000020000000100000040
000000020000000100000041
```

À la sortie du mode *recovery*, l'instance doit choisir une nouvelle *timeline*. Les *timelines* connues avec leur point de départ sont suivies grâce aux fichiers d'historique, nommés d'après le numéro hexadécimal sur huit caractères de la *timeline* et le suffixe `.history`, et archivés avec les journaux. En partant de la *timeline* qu'elle quitte, l'instance restaure les fichiers historiques des *timelines* suivantes pour choisir la première disponible, puis archive un nouveau fichier `.history` pour la nouvelle *timeline* sélectionnée, avec l'adresse du point de départ dans la *timeline* qu'elle quitte :

```
$ cat 00000002.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02
```

Après une seconde restauration, ciblant la *timeline* 2, l'instance choisit la *timeline* 3 :

```
$ cat 00000003.history
1   0/9765A80   before 2015-10-20 16:59:30.103317+02
2   0/105AF7D0  before 2015-10-22 10:25:56.614316+02
```

On peut choisir la *timeline* cible en configurant le paramètre `recovery_target_timeline`. `recovery_target_timeline` est par défaut à `latest`, et la restauration suit donc les changements de *timeline* depuis le moment de la sauvegarde.

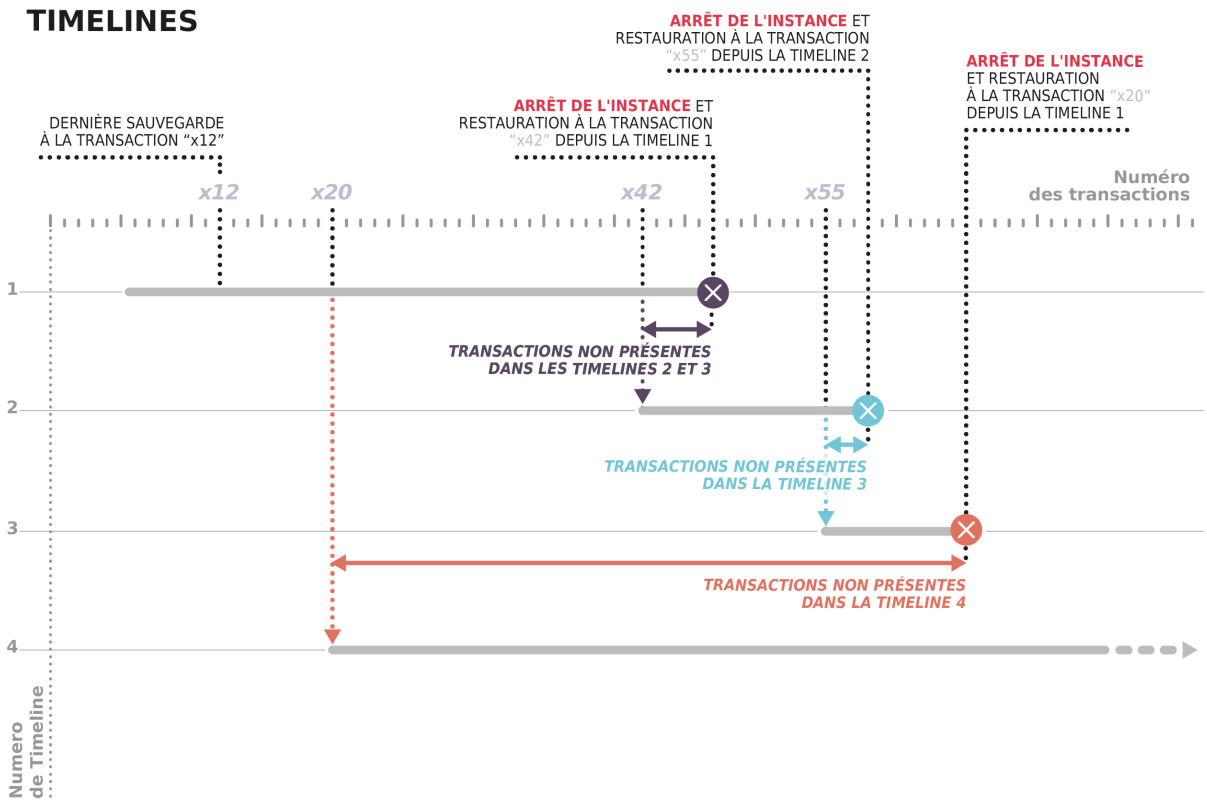
Pour choisir une autre *timeline* que la dernière, il faut donner le numéro de la *timeline* cible comme valeur du paramètre `recovery_target_timeline`. Les timelines permettent d'effectuer plusieurs restaurations successives à partir du même *base backup* et d'archiver au même endroit sans mélanger les journaux.

Bien sûr, pour restaurer dans une *timeline* précise, il faut que le fichier `.history` correspondant soit encore présent dans les archives, sous peine d'erreur.

Il y a quelques pièges :

Le numéro de timeline dans les traces ou affiché par `pg_controldata` est en décimal. Mais les fichiers `.history` portent un numéro en hexadécimal (par exemple `00000014.history` pour la timeline 20). On peut fournir les deux à `recovery_target_timeline` (`20` ou `'0x14'`). Attention, il n'y a pas de contrôle !
Et attention sur les anciennes versions : jusque PostgreSQL 11 compris, la valeur par défaut de `recovery_target_timeline` était `current` : la restauration se faisait donc dans la même *timeline* que le *base backup*. Si entre-temps il y avait eu une bascule ou une précédente restauration, la nouvelle timeline n'était pas automatiquement suivie !

2.6.8 Restauration PITR : illustration des timelines



Ce schéma illustre ce processus de plusieurs restaurations successives, et la création de différentes *timelines* qui en résulte.

On observe ici les éléments suivants avant la première restauration :

- la fin de la dernière sauvegarde se situe en haut à gauche sur l'axe des transactions, à la transaction `x12` ;
- cette sauvegarde a été effectuée alors que l'instance était en activité sur la *timeline* 1.

On décide d'arrêter l'instance alors qu'elle est arrivée à la transaction `x47`, par exemple parce qu'une nouvelle livraison de l'application a introduit un bug qui provoque des pertes de données. L'objectif est de restaurer l'instance avant l'apparition du problème afin de récupérer les données dans un état cohérent, et de relancer la production à partir de cet état. Pour cela, on restaure les fichiers de l'instance à partir de la dernière sauvegarde, puis on modifie le fichier de configuration pour que l'instance, lors de sa phase de *recovery* :

- restaure les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaure les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x42`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la

timeline 2, la bifurcation s'effectuant à la transaction `x42`. L'instance étant de nouveau ouverte en écriture, elle va générer de nouveaux WAL, qui seront associés à la nouvelle *timeline* : ils n'écrasent pas les fichiers WAL archivés de la *timeline* 1, ce qui permet de les réutiliser pour une autre restauration en cas de besoin (par exemple si la transaction `x42` utilisée comme point d'arrêt était trop loin dans le passé, et que l'on désire restaurer de nouveau jusqu'à un point plus récent).

Un peu plus tard, on a de nouveau besoin d'effectuer une restauration dans le passé - par exemple, une nouvelle livraison applicative a été effectuée, mais le bug rencontré précédemment n'était toujours pas corrigé. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, puis on configure PostgreSQL pour suivre la *timeline* 2 (paramètre `recovery_target_timeline = 2`) jusqu'à la transaction `x55`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point de la bifurcation (transaction `x42`) ;
- suivre la *timeline* indiquée (2) et rejouer les WAL archivés jusqu'au point précédant immédiatement l'apparition du bug applicatif (transaction `x55`).

On démarre ensuite l'instance et on l'ouvre en écriture, on constate alors que celle-ci bascule sur la *timeline* 3, la bifurcation s'effectuant cette fois à la transaction `x55`.

Enfin, on se rend compte qu'un problème bien plus ancien et subtil a été introduit précédemment aux deux restaurations effectuées. On décide alors de restaurer l'instance jusqu'à un point dans le temps situé bien avant, jusqu'à la transaction `x20`. On restaure donc de nouveau les fichiers de l'instance à partir de la même sauvegarde, et on configure le serveur pour restaurer jusqu'à la transaction `x20`. Lors du *recovery*, l'instance va :

- restaurer les WAL archivés jusqu'à l'état de cohérence (transaction `x12`) ;
- restaurer les WAL archivés jusqu'au point précédant immédiatement l' apparition du bug applicatif (transaction `x20`).

Comme la création des deux *timelines* précédentes est archivée dans les fichiers *history*, l'ouverture de l'instance en écriture va basculer sur une nouvelle *timeline* (4). Suite à cette restauration, toutes les modifications de données provoquées par des transactions effectuées sur la *timeline* 1 après la transaction `x20`, ainsi que celles effectuées sur les *timelines* 2 et 3, ne sont donc pas présentes dans l'instance.

2.6.9 Après la restauration

- Bien vérifier que l'archivage a repris
  - et que les archives des journaux sont complètes
- Ne pas hésiter à reprendre une sauvegarde complète
- Bien vérifier que les secondaires ont suivi

Une fois le nouveau primaire en place, la production peut reprendre, mais il faut vérifier que la sauvegarde PITR est elle aussi fonctionnelle.

Ce nouveau primaire a généralement commencé à archiver ses journaux à partir du dernier journal récupéré de l'ancien primaire, renommé avec l'extension `.partial`, juste avant la bascule sur la nouvelle *timeline*. Il faut bien sûr vérifier que l'archivage des nouveaux journaux fonctionne.

Sur l'ancien primaire, les derniers journaux générés juste avant l'incident n'ont pas forcément été archivés. Ceux-ci possèdent un fichier témoin `.ready` dans `pg_wal/archive_status`. Même s'ils ont été copiés manuellement vers le nouveau primaire avant sa promotion, celui-ci ne les a pas archivés.

Rappelons qu'un « trou » dans le flux des journaux dans le dépôt des archives empêchera la restauration d'aller au-delà de ce point !

Il est possible de forcer l'archivage des fichiers `.ready` depuis l'ancien primaire, avant la bascule, en exécutant à la main les `archive_command` que PostgreSQL aurait générées, mais la facilité pour le faire dépend de l'outil.

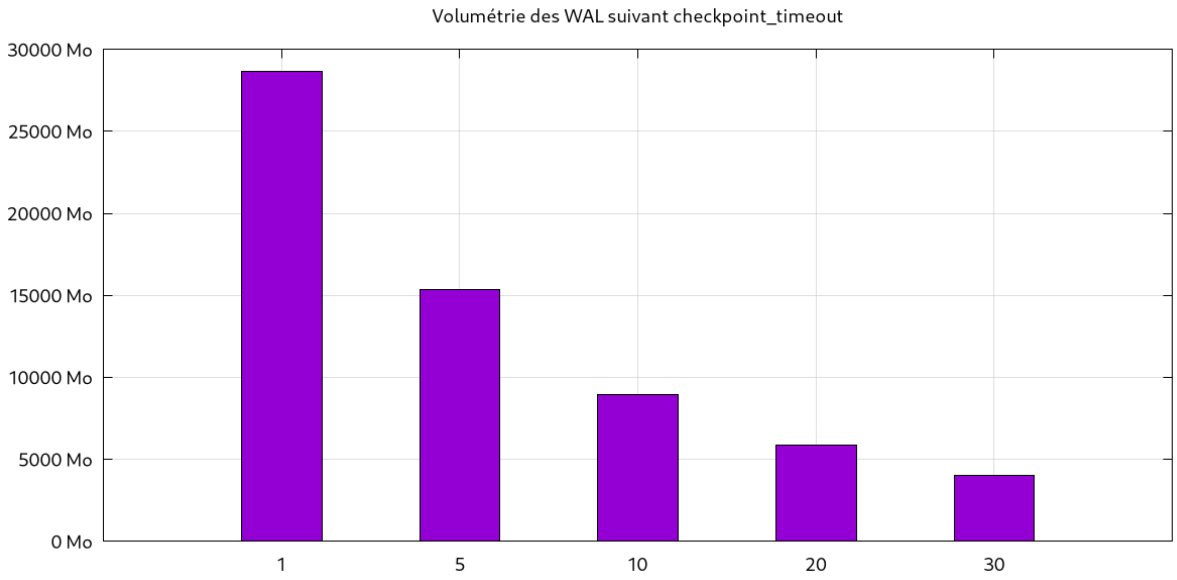
La copie de journaux à la main est donc risquée. Il ne faut pas hésiter à reprendre une sauvegarde complète du nouveau primaire pour repartir d'une base sûre.

Quant aux éventuelles instances secondaires, il est vraisemblable qu'elles doivent être reconstruites suite à la restauration de l'instance primaire. (Si elles ont appliqué des journaux qui ont été perdus et n'ont pas été repris par le primaire restauré, ces secondaires ne pourront se raccrocher. Consulter les traces.)

2.7 POUR ALLER PLUS LOIN

- Limiter la volumétrie des journaux sauvegardés
- Quels sont les outils PITR ?

2.7.1 Réduire le nombre de journaux sauvegardés



- Monter
  - `checkpoint_timeout`
  - `max_wal_size`

L'un des problèmes de la sauvegarde PITR est la place prise sur disque par les journaux de transactions. Si un journal de 16 Mo (par défaut) est généré toutes les minutes, le total est de 23 Go de journaux par jour, et parfois beaucoup plus. Il n'est pas forcément possible de conserver autant de journaux.

Un premier moyen est de reduire la volumétrie à la source en espaçant les checkpoints. Le graphique ci-dessus représente la volumétrie générée par un simple test avec `pgbench` (OLTP classique donc) avec `checkpoint_timeout` variant entre 1 et 30 minutes : les écarts sont énormes.

La raison est que, pour des raisons de fiabilité, un bloc modifié est intégralement écrit (8 ko) dans les journaux à sa première modification après un checkpoint. Par la suite, seules les modifications de ce bloc, souvent beaucoup plus petites, sont journalisées. (Ce comportement dépend du paramètre `full_page_writes`[11], activé par défaut et qu'il est impératif de laisser tel quel, sauf peut-être sur ZFS.)

Espacer les checkpoints permet d'économiser des écritures de blocs complets, si l'activité s'y prête (en OLTP surtout). Il y a un intérêt en performances, mais surtout en place disque économisée quand les journaux sont archivés, aussi accessoirement en CPU s'ils sont compressés, et en trafic réseau s'ils sont répliqués.

Par cohérence, si l'on monte `checkpoint_timeout`, il faut penser à augmenter aussi `max_wal_size`, et vice-versa. Des valeurs courantes sont respectivement 15 minutes, parfois plus, et plusieurs gigaoctets.

Il y a cependant un inconvénient : un écart plus grand entre checkpoints peut allonger la restauration après un arrêt brutal, car il y aura plus de journaux à rejouer, parfois des centaines ou des milliers.

2.7.2 Compresser les journaux de transactions

- `wal_compression`
  - moins de journaux
  - un peu plus de CPU
  - à activer
- Outils de compression standards : `gzip`, `bzip2`, `lzma`…
  - attention à ne pas ralentir l'archivage

PostgreSQL peut compresser les journaux à la source, si le paramètre `wal_compression` (désactivé par défaut) est passé à `on`. La compression est opérée par PostgreSQL au niveau de la page, avec un coût en CPU à l'écriture des journaux, très minime, et un gros gain en volumétrie (souvent plus de 50 % !). Comme il y a moins de journaux, leur rejeu est aussi plus rapide, ce qui accélère la réplication et la reprise après un crash. Le prix est une augmentation de la consommation en CPU. Les détails et un exemple figurent dans ce billet du blog Dalibo[12].

Une autre solution est la compression à la volée des journaux archivés dans l'`archive_command`. Les outils classiques comme `gzip`, `bzip2`, `lzma`, `xz`, etc. conviennent. Tous les outils PITR incluent plusieurs de ces algorithmes. Un fichier de 16 Mo aura généralement une taille compressée comprise entre 3 et 6 Mo.

[11] https://wiki.postgresql.org/wiki/Full_page_writes
[12] https://blog.dalibo.com/2024/01/05/cambouis.html

Cependant, attention au temps de compression des journaux : en cas d'écritures lourdes, une compression élevée mais lente peut mener à un retard conséquent de l'archivage par rapport à l'écriture des journaux, jusque saturation de `pg_wal`, et arrêt de l'instance. Il est courant de se contenter de `gzip -1` ou `lz4 -1` pour les journaux, et de ne compresser agressivement que les sauvegardes des fichiers de la base.

2.7.3 Outils de sauvegarde PITR dédiés

- Se faciliter la vie avec différents outils
  - pgBackRest
  - barman
  - pitrery (< v15, déprécié)
- Fournissent :
  - un outil pour les backups, les purges…
  - une commande pour l'archivage

Il n'est pas conseillé de réinventer la roue et d'écrire soi-même des scripts de sauvegarde, qui doivent prévoir de nombreux cas et bien gérer les erreurs. La sauvegarde concurrente est également difficile à manier. Des outils reconnus existent, dont nous évoquerons brièvement les plus connus. Il en existe d'autres. Ils ne font pas partie du projet PostgreSQL à proprement parler et doivent être installés séparément.

Les outils décrits succinctement plus bas fournissent :

- un outil pour procéder aux sauvegardes, gérer la péremption des archives… ;
- un outil qui sera appelé par `archive_command`.

Leur philosophie peut différer, notamment en terme de centralisation ou de compromis entre simplicité et fonctionnalités. Ces dernières s'enrichissent d'ailleurs au fil du temps.

2.7.4 pgBackRest

- Gère la sauvegarde et la restauration
  - *pull* ou *push*, multidépôts
  - mono ou multi-serveurs
- Indépendant des commandes système
  - protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Multi-thread, sauvegarde depuis un secondaire, archivage asynchrone…
- Projet mature

pgBackRest[13] est un outil de gestion de sauvegardes PITR écrit en perl et en C, par David Steele de Crunchy Data.

Il met l'accent sur les performances avec de gros volumes et les fonctionnalités, au prix d'une complexité à la configuration :

- un protocole dédié pour le transfert et la compression des données ;
- des opérations parallélisables en multi-thread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité d'archiver ou restaurer les WAL de façon asynchrone, et donc plus rapide ;
- la possibilité d'abandonner l'archivage en cas d'accumulation et de risque de saturation de `pg_wal` ;
- la gestion de dépôts de sauvegarde multiples (pour sécuriser, ou avoir plusieurs niveaux d'archives) ;
- le support intégré de dépôts S3 ou Azure ;
- le support d'un accès TLS géré par pgBackRest en alternative à SSH ;
- la sauvegarde depuis un serveur secondaire ;
- le chiffrement des sauvegardes ;
- la restauration en mode delta, très pratique pour restaurer un serveur qui a décroché mais n'a que peu divergé ;
- la reprise d'une sauvegarde échouée.

pgBackRest n'utilise pas `pg_receivewal` pour garantir la sauvegarde du dernier journal (non terminé) avant un sinistre. Les auteurs considèrent que dans ce cas un secondaire synchrone est plus adapté et plus fiable.

Le projet est très actif et considéré comme fiable, et les fonctionnalités proposées sont intéressantes.

[13] https://pgbackrest.org/

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest[14].

2.7.5 barman

- Gère la sauvegarde et la restauration
  - mode *pull*
  - multi-serveurs
- Une seule commande (`barman`)
- Et de nombreuses actions
  - `list-server`, `backup`, `list-backup`, `recover`…
- Spécificité : gestion de `pg_receivewal`

barman est un outil créé par 2ndQuadrant (racheté depuis par EDB). Il a pour but de faciliter la mise en place de sauvegardes PITR. Il gère à la fois la sauvegarde et la restauration.

La commande `barman` dispose de plusieurs actions :

- `list-server`, pour connaître la liste des serveurs configurés ;
- `backup`, pour lancer une sauvegarde de base ;
- `list-backup`, pour connaître la liste des sauvegardes de base ;
- `show-backup`, pour afficher des informations sur une sauvegarde ;
- `delete`, pour supprimer une sauvegarde ;
- `recover`, pour restaurer une sauvegarde (la restauration peut se faire à distance).

Contrairement aux autre outils présentés ici, barman permet d'utiliser `pg_receivewal`.

Il supporte aussi les dépôts S3 ou blob Azure.

Site web de barman[15]

[14] https://github.com/pgstef/check_pgbackrest/
[15] https://www.pgbarman.org/

2.7.6 pitrery

- Projet en fin de vie, non compatible v15+
- Gère la sauvegarde et la restauration
  - mode push
  - mono-serveur
- Multi-commandes
  - `archive_wal`
  - `pitrery`
  - `restore_wal`

pitrery a été créé par la société Dalibo. Il mettait l'accent sur la simplicité de sauvegarde et la restauration de la base.

Après 10 ans de développement actif, le projet Pitrery est désormais placé en maintenance LTS (*Long Term Support*) jusqu'en novembre 2026. Plus aucune nouvelle fonctionnalité n'y sera ajoutée, les mises à jour concerneront les correctifs de sécurité uniquement. Il est désormais conseillé de lui préférer pgBackRest. Il n'est plus compatible avec PostgreSQL 15 et supérieur.

Site Web de pitrery[16].

[16] https://dalibo.github.io/pitrery/

2.8 CONCLUSION

- Une sauvegarde
  - fiable
  - éprouvée
  - rapide
  - continue
- Mais
  - plus complexe à mettre en place que `pg_dump`
  - qui restaure toute l'instance

Cette méthode de sauvegarde est la seule utilisable dès que les besoins de performance de sauvegarde et de restauration augmentent (*Recovery Time Objective* ou RTO), ou que le volume de perte de données doit être drastiquement réduit (*Recovery Point Objective* ou RPO).

2.8.1 Questions

N'hésitez pas, c'est le moment !

2.9 QUIZ

https://dali.bo/i2_quiz

2.10 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

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

Elle ne dure que quelques minutes.

2.10.1 Sur Rocky Linux 8 ou 9

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

Installation du dépôt communautaire :

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

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

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

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

# dnf -qy module disable postgresql
```

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

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

Les outils clients et les librairies nécessaires seront automatiquement installés.

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

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

Création d'une première instance :

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

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

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

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

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

Chemins :

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

Configuration :

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

Commandes d'administration habituelles :

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

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

Test rapide de bon fonctionnement et connexion à psql :

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

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

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

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

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

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

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

Création d'autres instances :

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

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

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

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

- Création du fichier service de la deuxième instance :

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

- Modification de ce dernier fichier avec le nouveau chemin :

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

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

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

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

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

- Ouvrir le nouveau port dans le firewall au besoin.

2.10.2 Sur Debian / Ubuntu

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

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

Installation du dépôt communautaire :

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

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

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

Installation de PostgreSQL 16 :

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

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

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

Puis installer les paquets serveur et clients et leurs dépendances :

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

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

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

Chemins :

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

Configuration

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

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

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

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

Démarrage de l'instance avec le serveur :

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

Ouverture du firewall :

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

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

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

Destruction d'une instance :

```
# pg_dropcluster 16 main
```

Création d'autres instances :

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

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

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

- créer une instance :

```
# pg_createcluster 16 infocentre
```

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

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

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

```
# pg_ctlcluster 16 infocentre start
```

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

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

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

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

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

```
local   all             postgres                                trust
```

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

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

Une authentification par mot de passe est plus sécurisée :

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

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

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

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

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

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

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

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

Ce fichier doit être protégé des autres utilisateurs :

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

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

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

Rappels :

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

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

root:~# pg_ctlcluster 16 main reload

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

2.11 INTRODUCTION À PGBENCH

pgbench est un outil de test livré avec PostgreSQL. Son but est de faciliter la mise en place de benchmarks simples et rapides. Par défaut, il installe une base assez simple, génère une activité plus ou moins intense et calcule le nombre de transactions par seconde et la latence. C'est ce qui sera fait ici dans cette introduction. On peut aussi lui fournir ses propres scripts.

La documentation complète est sur https://docs.postgresql.fr/current/pgbench.html. L'auteur principal, Fabien Coelho, a fait une présentation complète, en français, à la PG Session #9 de 2017[17].

2.11.1 Installation

L'outil est installé avec les paquets habituels de PostgreSQL, client ou serveur suivant la distribution.

Dans le cas des paquets RPM du PGDG, l'outil n'est pas dans le PATH par défaut ; il faudra donc fournir le chemin complet :

```
/usr/pgsql-16/bin/pgbench
```

Il est préférable de créer un rôle non privilégié dédié, qui possédera la base de données :

```
CREATE ROLE pgbench LOGIN PASSWORD 'unmotdepassebienc0mplexe';
CREATE DATABASE pgbench OWNER pgbench ;
```

Le `pg_hba.conf` doit éventuellement être adapté.

La base par défaut s'installe ainsi (indiquer la base de données en dernier ; ajouter `-p` et `-h` au besoin) :

```
pgbench -U pgbench --initialize --scale=100 pgbench
```

`--scale` permet de faire varier proportionnellement la taille de la base. À 100, la base pèsera 1,5 Go, avec 10 millions de lignes dans la table principale `pgbench_accounts` :

```
pgbench@pgbench=# \d+
                          Liste des relations
 Schéma |       Nom        | Type  | Propriétaire | Taille  | Description
--------+------------------+-------+--------------+---------+-------------
 public | pg_buffercache   | vue   | postgres     | 0 bytes |
 public | pgbench_accounts | table | pgbench      | 1281 MB |
 public | pgbench_branches | table | pgbench      | 40 kB   |
 public | pgbench_history  | table | pgbench      | 0 bytes |
 public | pgbench_tellers  | table | pgbench      | 80 kB   |
```

2.11.2 Générer de l'activité

Pour simuler une activité de 20 clients simultanés, répartis sur 4 processeurs, pendant 100 secondes :

[17] https://youtu.be/aTwh_CgRaE0

```
pgbench -U pgbench -c 20 -j 4 -T100 pgbench
```

NB : ne **pas** utiliser `-d` pour indiquer la base, qui signifie `--debug` pour pgbench, qui noiera alors l'affichage avec ses requêtes :

```
 UPDATE pgbench_accounts SET abalance = abalance + -3455 WHERE aid = 3789437;
 SELECT abalance FROM pgbench_accounts WHERE aid = 3789437;
 UPDATE pgbench_tellers SET tbalance = tbalance + -3455 WHERE tid = 134;
 UPDATE pgbench_branches SET bbalance = bbalance + -3455 WHERE bid = 78;
 INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
  VALUES (134, 78, 3789437, -3455, CURRENT_TIMESTAMP);
```

À la fin, s'affichent notamment le nombre de transactions (avec et sans le temps de connexion) et la durée moyenne d'exécution du point de vue du client (`latency`) :

```
 scaling factor: 100
 query mode: simple
 number of clients: 20
 number of threads: 4
 duration: 10 s
 number of transactions actually processed: 20433
 latency average = 9.826 ms
 tps = 2035.338395 (including connections establishing)
 tps = 2037.198912 (excluding connections establishing)
```

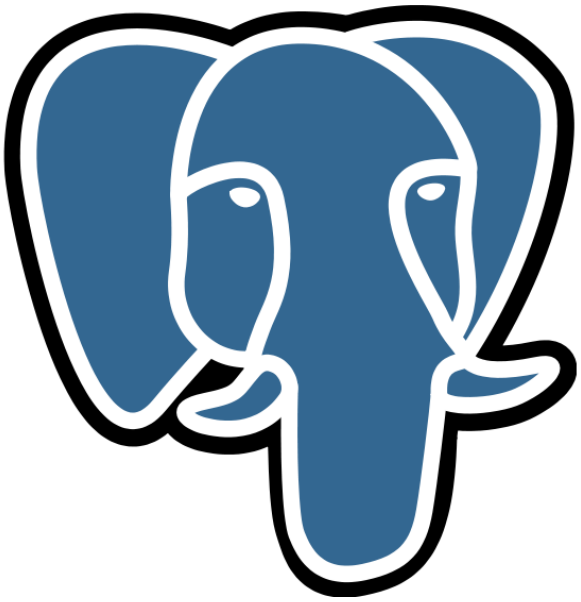
Modifier le paramétrage est facile grâce à la variable d'environnement `PGOPTIONS` :

```
PGOPTIONS='-c synchronous_commit=off -c commit_siblings=20' \
             pgbench -d pgbench -U pgbench -c 20 -j 4 -T100   2>/dev/null

latency average = 6.992 ms
tps = 2860.465176 (including connections establishing)
tps = 2862.964803 (excluding connections establishing)
```

Des tests rigoureux doivent durer bien sûr beaucoup plus longtemps que 100 s, par exemple pour tenir compte des effets de cache, des checkpoints périodiques, etc.

3/ PostgreSQL : Outils de sauvegarde physique



3.1 INTRODUCTION

- 2 mécanismes de sauvegarde natifs et robustes
- Industrialisation fastidieuse
- Des outils existent !!

Nous avons vu le fonctionnement interne du mécanisme de sauvegarde physique. Celui-ci étant en place nativement dans le moteur PostgreSQL depuis de nombreuses versions, sa robustesse n'est plus à prouver. Cependant, son industrialisation reste fastidieuse.

Des outils tiers existent et vont permettre de faciliter la gestion des sauvegardes, de leur mise en place jusqu'à la restauration. Dans ce module nous allons voir en détail certains de ces outils et étudier les critères qui vont nous permettre de choisir la meilleure solution selon notre contexte.

3.1.1 Au menu

- Présentation:
  - pg_basebackup
  - pgBackRest
  - Barman
- Comment choisir ?

Lors de cette présentation, nous allons passer en revue les différents outils principaux de gestion de sauvegardes, leurs forces, le paramétrage, l'installation et l'exploitation.

3.1.2 Définition du besoin - Contexte

- Sauvegarde locale (ex. NFS) ?
- Copie vers un serveur tiers (push) ?
- Sauvegarde distante initiée depuis un serveur tiers (pull) ?
- Ressources à disposition ?
- Accès SSH ?
- OS ?
- Sauvegardes physiques ? Logiques ?
- Version de PostgreSQL ?
- Politique de rétention ?

Où les sauvegardes doivent-elles être stockées ?

Quelles ressources sont à disposition : serveur de sauvegarde dédié ? quelle puissance pour la compression ?

De quel type d'accès aux serveurs de base de données dispose-t-on ? Quelle est la version du système d'exploitation ?

Il est très important de se poser toutes ces questions, les réponses vont servir à établir le contexte et permettre de choisir l'outil et la méthode la plus appropriée.

Attention, pour des raisons de sécurité et de fiabilité, les répertoires choisis pour la restauration des données de votre instance **ne doivent pas** être à la racine d'un point de montage.
Si un ou plusieurs points de montage sont dédiés à l'utilisation de PostgreSQL, positionnez toujours les données dans un sous-répertoire, voire deux niveaux en dessous du point de montage (eg. `<point de montage>/<version majeure>/<nom instance>`).

3.2 PG_BASEBACKUP - PRÉSENTATION

- Outil intégré à PostgreSQL
- Prévu pour créer une instance secondaire
- Pour sauvegarde ponctuelle
  - PITR avec outils complémentaires

pg_basebackup[1] est une application cliente intégrée à PostgreSQL, au même titre que pg_dump ou pg_dumpall.

pg_basebackup a été conçu pour permettre l'initialisation d'une instance secondaire, et il peut donc être utilisé pour effectuer facilement une sauvegarde physique ponctuelle. Celle-ci inclut les fichiers et journaux nécessaires pour une restauration telle que l'instance était à la fin de la sauvegarde.

pg_basebackup peut aussi être à la base d'outils permettant le PITR (par exemple barman). Ces outils s'occupent en plus de l'archivage des journaux générés pendant et après la sauvegarde initiale, pour une restauration dans un état postérieur à la fin de cette sauvegarde.

3.2.1 pg_basebackup - Formats de sauvegarde

- `--format plain`
  - arborescence identique à l'instance sauvegardée
- `--format tar`
  - archive
  - compression : `-z`, `-Z (0..9)`

Le format par défaut de la sauvegarde est `plain`, ce qui signifie que les fichiers seront créés tels quels dans le répertoire de destination (ou les répertoires en cas de tablespaces). C'est idéal pour obtenir une copie immédiatement utilisable.

Pour une archive à proprement parler, préférer l'option `--format tar`. pg_basebackup génère alors une archive `base.tar` pour le PGDATA de l'instance, puis une archive `<oid>.tar` par tablespace.

[1] https://www.postgresql.org/docs/current/static/app-pgbasebackup.html

Les journaux récupérés seront également dans un fichier `.tar`.

L'option `--gzip` (`-z`) ajoute la compression `gzip`. Le niveau de compression peut également être spécifié avec `--compress=1` à `9` (`-Z`). Cela permet d'arbitrer entre la durée de la sauvegarde et sa taille.

3.2.2 pg_basebackup - Avantages

- Transfert des WAL pendant la sauvegarde
- Slot de réplication automatique (temporaire voire permanent)
- Limitation du débit
- Relocalisation des tablespaces
- Fichier manifeste (v13+)
- Vérification des checksums
- Sauvegarde possible à partir d'un secondaire
- Compression côté serveur ou client (v15+)
- Emplacement de la sauvegarde (client/server/blackhole) (v15+)
- Suivi : `pg_stat_progress_basebackup` (v13+)

pg_basebackup s'est beaucoup amélioré au fil des versions et son comportement a parfois changé. Regardez bien la documentation[2] de votre version.

Même avec un serveur un peu ancien, il possible d'installer un `pg_basebackup` récent, en installant les outils clients de la dernière version de PostgreSQL.

Récupération des journaux :

pg_basebackup sait récupérer les fichiers WAL nécessaires à la restauration de la sauvegarde sans passer par la commande d'archivage. Il connaît deux méthodes :

Avec l'option `--wal-method fetch` (ou `-X`), les WAL générés pendant la sauvegarde seront demandés une fois celle-ci terminée, à condition qu'ils n'aient pas été recyclés entre-temps (ce qui peut nécessiter un slot de réplication, ou éventuellement une configuration élevée du paramètre `wal_keep_size` / `wal_keep_segments`).

L'option par défaut est cependant `-X stream` : les WAL sont récupérés non pas en fin de sauvegarde, mais *en streaming* pendant celle-ci. Cela nécessite néanmoins l'utilisation d'un *wal sender* supplémentaire, le paramètre `max_wal_senders` doit parfois être augmenté en conséquence.

[2] https://docs.postgresql.fr/current/app-pgbasebackup.html

Rappelons que si l'archivage des WAL n'est pas actif, la sauvegarde effectuée ne sera utilisée que pour restaurer l'instance telle qu'elle était au moment de la fin de la sauvegarde : il ne sera pas possible de réaliser une restauration PITR.

À l'inverse, `-X none` peut être utile si la récupération des journaux est réalisée par ailleurs (généralement par `archive_command` ou `archive_library`). Attention, l'archive réalisée avec pg_basebackup ne sera alors pas « complète », et ne pourra pas être restaurée sans ces archives des journaux (il faudra indiquer où aller les chercher avec `restore_command`.)

Slots de réplication :

Par défaut, pg_basebackup va créer un slot de réplication temporaire sur le serveur pour sécuriser la sauvegarde. Il disparaîtra une fois celle-ci terminée.

Pour faciliter la mise en place d'une instance secondaire, et garantir que tous les journaux nécessaires seront encore sur le primaire à son démarrage, il est possible de créer un slot de réplication permanent, et de le fournir à pg_basebackup avec `--slot nom_du_slot`. pg_basebackup peut le créer lui-même avec `--create`. Si l'on préfère le créer préalablement, il suffit d'exécuter la requête suivante :

```
SELECT pg_create_physical_replication_slot ('nom_du_slot');
```

Rappelons qu'un slot initialisé mais inutilisé doit être rapidement supprimé pour ne pas mener à une dangereuse accumulation des journaux.

Sécurisation de la sauvegarde :

Par défaut, `pg_basebackup` crée un fichier manifeste (à partir de PostgreSQL 13). Ce fichier contient la liste des fichiers sauvegardés, leur taille et leur somme de contrôle. Cela permet après coup de vérifier l'intégrité de la sauvegarde à l'aide de l'outil `pg_verifybackup`.

L'algorithme par défaut de la somme de contrôle, CRC32, suffit pour détecter une erreur technique accidentelle ; d'autres algorithmes disponibles permettent de détecter une manipulation volontaire de la sauvegarde.

Vérification des sommes de contrôle :

Une sauvegarde avec pg_basebackup entraîne la vérification des sommes de contrôle de l'instance. Cela garantit que la sauvegarde n'héritera pas d'une corruption existante, sinon l'outil tombe en erreur.

L'option `--no-verify-checksums` autorise la sauvegarde d'une instance où une corruption est détectée (sauvegarde aussi problématique, certes, mais qui peut permettre de travailler sur la récupération, ou de sauver l'essentiel).

Emplacement de la sauvegarde

À partir de la version 15, l'option `--target` permet de spécifier où la sauvegarde doit être réalisée :

- sur le serveur où la commande est lancée (`client`) ;
- sur le serveur de base de données (`server`) ;
- dans le vide (`blackhole`).

Des destinations peuvent être ajoutées par des extensions, `basebackup_to_shell`[3] est fournie à titre d'exemple et permet d'exécuter une commande à l'issue d'une sauvegarde.

Lorsque la destination `server` est choisie, plusieurs restrictions s'appliquent à la sauvegarde :

- le format doit être `tar` ;
- l'utilisateur employé pour la réaliser doit être membre du rôle `pg_write_server_files` ;
- la méthode de récupération des WAL doit être `fetch` ou `none`.

Compression de la sauvegarde :

À partir de la version 15, il est possible de demander la compression de la sauvegarde avec un grand niveau de personnalisation :

- algorithme de compression parmi `gzip`, `lz4` et `zstd` ;
- rapidité de la compression hors parallélisme (`lz4`) ;
- niveau de compression (`zstd`) ;
- parallélisation de la compression (`zstd`) ;
- localisation de la compression (serveur ou client).

Cela permet de gérer différents scénarios et d'éviter certains goulets d'étranglement lors d'une sauvegarde.

Autres options :

Le débit de la sauvegarde est configurable avec l'option `--max-rate=` (`-r`) pour limiter l'impact sur l'instance ou le réseau. Cette restriction de débit ne concerne pas les journaux transférés en parallèle (`-X stream`).

Pour gagner un peu de temps, si l'instance n'est pas trop chargée, `--checkpoint=fast` accélère le checkpoint préalable à la sauvegarde.

Avec une sauvegarde `plain`, il est possible de modifier sur la cible les chemins des éventuels tablespaces avec l'option `--tablespace-mapping=<vieuxrep>=<nouveaurep>` (ou `-T`), et de relocaliser le répertoire des fichiers WAL avec l'option `--waldir=<nouveau chemin>`.

Depuis un secondaire :

pg_basebackup permet nativement de réaliser une sauvegarde à partir d'une instance secondaire. Le paramétrage nécessaire figure plus bas.

Suivi :

Pour suivre le déroulement de la sauvegarde depuis un terminal, il existe l'option `--progress` (`-P`).

À partir de PostgreSQL 13, il existe aussi une vue pour ce suivi : `pg_stat_progress_basebackup`[4].

Options complètes :

[3] https://docs.postgresql.fr/current/basebackup-to-shell.html
[4] https://docs.postgresql.fr/current/progress-reporting.html#BASEBACKUP-PROGRESS-REPORTING

Pour mémoire, toutes les options disponibles sont celles-ci (en version 15) :

```
$ pg_basebackup --help

pg_basebackup prend une sauvegarde binaire d'un serveur PostgreSQL en cours
d'exécution.

Usage :
  pg_basebackup [OPTION]...

Options contrôlant la sortie :
  -D, --pgdata=RÉPERTOIRE        reçoit la sauvegarde de base dans ce répertoire
  -F, --format=p|t               format en sortie (plain (par défaut), tar)
  -r, --max-rate=TAUX            taux maximum de transfert du répertoire de
                                 données (en Ko/s, ou utiliser le suffixe « k »
                                 ou « M »)
  -R, --write-recovery-conf      écrit la configuration pour la réplication
  -t, --target=CIBLE[:DETAIL]    cible de sauvegarde (si autre que client)
  -T, --tablespace-mapping=ANCIENREP=NOUVEAUREP
                                 déplace le répertoire ANCIENREP en NOUVEAUREP
      --waldir=RÉP_WAL           emplacement du répertoire des journaux de
                                 transactions
  -X, --wal-method=none|fetch|stream
                                 inclut les journaux de transactions requis avec
                                 la méthode spécifiée
  -z, --gzip                     compresse la sortie tar
  -Z, --compress=[{client|server}-]METHODE[:DETAIL]
                                 compresse sur le client ou le serveur comme indiqué
  -Z, --compress=none            ne compresse pas la sortie tar

Options générales :
  -c, --checkpoint=fast|spread   exécute un CHECKPOINT rapide ou réparti
      --create-slot              crée un slot de réplication
  -l, --label=LABEL              configure le label de sauvegarde
  -n, --no-clean                 ne nettoie pas en cas d'erreur
  -N, --no-sync                  n'attend pas que les modifications soient
                                 proprement écrites sur disque
  -P, --progress                 affiche la progression de la sauvegarde
  -S, --slot=NOMREP              slot de réplication à utiliser
  -v, --verbose                  affiche des messages verbeux
  -V, --version                  affiche la version puis quitte
      --manifest-checksums=SHA{224,256,384,512}|CRC32C|NONE
                                 utilise cet algorithme pour les sommes de
                                 contrôle du manifeste
      --manifest-force-encode    encode tous les noms de fichier dans le
                                 manifeste en hexadécimal
      --no-estimate-size         ne réalise pas d'estimation sur la taille de la
                                 sauvegarde côté serveur
      --no-manifest              supprime la génération de manifeste de
                                 sauvegarde
      --no-slot                  empêche la création de slots de réplication
                                 temporaires
      --no-verify-checksums      ne vérifie pas les sommes de contrôle
  -?, --help                     affiche cette aide puis quitte

Options de connexion :
```

```
  -d, --dbname=CHAÎNE_CONNEX     chaîne de connexion
  -h, --host=HÔTE                hôte du serveur de bases de données ou
                                 répertoire des sockets
  -p, --port=PORT                numéro de port du serveur de bases de données
  -s, --status-interval=INTERVAL durée entre l'envoi de paquets de statut au
                                 serveur (en secondes)
  -U, --username=UTILISATEUR     se connecte avec cet utilisateur
  -w, --no-password              ne demande jamais le mot de passe
  -W, --password                 force la demande du mot de passe (devrait
                                 survenir automatiquement)

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>.
Page d'accueil de PostgreSQL : <https://www.postgresql.org/>
```

3.2.3 pg_basebackup - Limitations

- Configuration *streaming* nécessaire
- Pas de configuration de l'archivage
- Pas d'association WAL archivés / sauvegarde
- Pas de politique de rétention
  - sauvegarde ponctuelle
- Pas de gestion de la restauration !
  - manuel : `recovery.signal`, `restore_command`…
  - pour un secondaire : `--write-recovery-conf`

Configuration :

pg_basebackup étant conçu pour la mise en place d'une instance en réplication, l'instance principale nécessite d'être configurée en conséquence :

- `max_wal_senders` doit avoir une valeur supérieure à `0` pour permettre à pg_basebackup de se connecter (au moins `2` si on utilise le transfert des WAL par streaming) — c'est le cas par défaut ;
- le fichier `pg_hba.conf` de l'instance principale doit être configuré pour autoriser les connexions de type `replication` depuis la machine où la sauvegarde est déclenchée, par exemple ainsi :

```
host  replication  repli_user  192.168.0.100/32  scram-sha-256
```

Dans l'idéal, l'utilisateur employé est dédié à la réplication. Pour automatiser, stocker le mot de passe nécessaire dans un fichier `.pgpass`.

L'archivage n'est pas géré par pg_basebackup. Il ne récupère par *streaming* que les journaux nécessaires à la cohérence de sa sauvegarde. Il faudra paramétrer `archive_command` ou `archive_library` à la main pour une sauvegarde PITR.

Si la sauvegarde est effectuée à partir d'une instance secondaire :

- ces paramétrages sont nécessaires (et en place par défaut) :
  - instance secondaire ouverte en lecture (`hot_standby` à `on`) ;
  - `max_wal_senders` supérieur `0` et droits en place pour permettre à pg_basebackup de se connecter ;
  - écriture complète des pages dans les WAL activée (`full_page_writes` à `on`) ;
- pour du PITR :
  - l'archivage des fichiers WAL doit être configuré indépendamment.
  - une attention particulière doit être apportée au fait que tous les fichiers WAL nécessaires à la restauration ont bien été archivés.

Gestion des sauvegardes :

La gestion des sauvegardes (rétention, purge…) n'est pas prévue dans l'outil.

pg_basebackup n'effectue pas non plus de lien entre les WAL archivés et les sauvegardes effectuées (si pg_basebackup ne les sauvegarde pas lui-même avec l'option `-X`).

Restauration :

pg_basebackup n'offre pas d'outil ni d'option pour la restauration.

La copie est directement utilisable, éventuellement après déplacement et/ou décompression des `.tar.gz` . Mais, généralement, on ajoutera un fichier `recovery.signal` , et on définira la `restore_command` pour récupérer les archives. Dans l'idéal, `restore_command` sera déjà prête dans le `postgresql.conf` .

Si le but est de monter un serveur secondaire de l'instance copiée, il existe une option utile : `--write-recovery-conf` (ou `-R`), qui génère la configuration nécessaire dans le répertoire de la sauvegarde (`postgresql.auto.conf` et fichier vide `standby.signal`). avec les paramètres pour une réplication en *streaming*.

3.3 PGBACKREST - PRÉSENTATION GÉNÉRALE

- David Steele (Crunchy Data)
- Langage : **C**
- License : **MIT** (libre)
- Type d'interface : **CLI** (ligne de commande)

3.3.1 pgBackRest - Fonctionnalités

- Gère la sauvegarde et la restauration
  - *pull* ou *push*, multidépôts
  - mono ou multi-serveurs
- Indépendant des commandes système
  - protocole dédié
- Sauvegardes complètes, différentielles ou incrémentales
- Multi-thread, sauvegarde depuis un secondaire, archivage asynchrone…
- Projet mature

pgBackRest[5] est un outil de gestion de sauvegardes PITR écrit en perl et en C, par David Steele de Crunchy Data.

Il met l'accent sur les performances avec de gros volumes et les fonctionnalités, au prix d'une complexité à la configuration :

- un protocole dédié pour le transfert et la compression des données ;
- des opérations parallélisables en multi-thread ;
- la possibilité de réaliser des sauvegardes complètes, différentielles et incrémentielles ;
- la possibilité d'archiver ou restaurer les WAL de façon asynchrone, et donc plus rapide ;
- la possibilité d'abandonner l'archivage en cas d'accumulation et de risque de saturation de `pg_wal` ;
- la gestion de dépôts de sauvegarde multiples (pour sécuriser, ou avoir plusieurs niveaux d'archives) ;
- le support intégré de dépôts S3 ou Azure ;

[5] https://pgbackrest.org/

- le support d'un accès TLS géré par pgBackRest en alternative à SSH ;
- la sauvegarde depuis un serveur secondaire ;
- le chiffrement des sauvegardes ;
- la restauration en mode delta, très pratique pour restaurer un serveur qui a décroché mais n'a que peu divergé ;
- la reprise d'une sauvegarde échouée.

pgBackRest n'utilise pas `pg_receivewal` pour garantir la sauvegarde du dernier journal (non terminé) avant un sinistre. Les auteurs considèrent que dans ce cas un secondaire synchrone est plus adapté et plus fiable.

Le projet est très actif et considéré comme fiable, et les fonctionnalités proposées sont intéressantes.

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest[6].

3.3.2 pgBackRest - Sauvegardes

- Type de sauvegarde : **physique/PITR** (à chaud)
- Type de stockage : **local**, **push** ou **pull**
- Planification : **crontab**
- Complètes, différentielles et incrémentales
- Compression des WAL

pgBackRest gère uniquement des sauvegardes physiques.

Il peut fonctionner soit en local (directement sur le serveur hébergeant l'instance à sauvegarder) pour un stockage local des sauvegardes, soit être exécuté depuis un serveur distant, déléguant ainsi l'ordonnancement, la compression et le stockage des données à celui-ci.

La technique utilisée pour la prise de sauvegarde repose sur le mécanisme interne standard et historique : `pg_backup_start()`, copie des fichiers, `pg_backup_stop()`.

3.3.3 pgBackRest - Restauration

- Depuis le serveur de BDD avec un dépôt local ou à distance
- Point dans le temps : date, identifiant de transaction, timeline ou point de restauration

[6] https://github.com/pgstef/check_pgbackrest/

La restauration d'une sauvegarde peut se faire soit localement, si les sauvegardes sont stockées en local, soit à distance. Dans ce dernier cas, les données à restaurer seront transférées via SSH.

Plusieurs types de point dans le temps peuvent être utilisés comme cible :

- la date ;
- un identifiant de transaction ;
- une timeline (en cas de divergence de timeline, `pgBackRest` peut restaurer les transactions issues d'une timeline précise) ;
- un point de restauration créé par un appel préalable à la fonction :
  - `pg_create_restore_point()`.

3.3.4 pgBackRest - Installation

- Accéder au dépôt communautaire PGDG
- Installer le paquet `pgbackrest`

pgBackRest est disponible sur le dépôt communautaire maintenu par la communauté PostgreSQL pour les systèmes d'exploitation disposant des gestionnaires de paquet au format deb (Debian, Ubuntu…)[7] ou rpm (Red Hat, Rocky Linux, CentOS, Fedora…)[8].

Il est recommandé de manière générale de privilégier une installation à partir de ces paquets plutôt que par les sources, essentiellement pour des raisons de maintenance.

[7] https://apt.postgresql.org/pub/repos/apt/
[8] https://yum.postgresql.org/

3.3.5 pgBackRest - Utilisation

```
Usage:
    pgbackrest [options] [command]

Commands:
    annotate        Add or modify backup annotation.
    archive-get     Get a WAL segment from the archive.
    archive-push    Push a WAL segment to the archive.
    backup          Backup a database cluster.
    check           Check the configuration.
    expire          Expire backups that exceed retention.
    help            Get help.
    info            Retrieve information about backups.
    repo-get        Get a file from a repository.
    repo-ls         List files in a repository.
    restore         Restore a database cluster.
    server          pgBackRest server.
    server-ping     Ping pgBackRest server.
    stanza-create   Create the required stanza data.
    stanza-delete   Delete a stanza.
    stanza-upgrade  Upgrade a stanza.
    start           Allow pgBackRest processes to run.
    stop            Stop pgBackRest processes from running.
    verify          Verify contents of the repository.
    version         Get version.
```

pgBackRest propose différentes commandes pouvant être passées en argument afin de contrôler les actions.

L'usage de ces différentes commandes sera détaillé ultérieurement.

3.3.6 pgBackRest - Configuration

- `/etc/pgbackrest.conf`
- Configuration générale dans la section `[global]`
- Chaque instance à sauvegarder doit avoir sa propre section, appelée `stanza`
- possibilité d'éclater la configuration dans plusieurs fichiers : `config-include-path`

Le format de configuration `INI` permet de définir des sections, qui sont matérialisées sous la forme d'une ligne : `[nomdesection]`.

pgBackRest s'attend à lire un fichier de configuration contenant la section `[global]`, contenant les paramètres de configuration globaux, et une section par instance à sauvegarder.

pgBackRest utilise le terme `stanza` pour regrouper l'ensemble des configurations à appliquer pour une **instance** à sauvegarder.

Exemple de configuration :

```
[global]
repo1-path=/var/lib/pgsql/10/backups

[erp_prod]
pg1-path=/var/lib/pgsql/10/data
```

Il peut y avoir plusieurs stanzas déclarées dans le fichier, notamment s'il est situé sur le serveur où sont stockées les sauvegardes de plusieurs instances.

Pour des questions de lisibilité, il est possible de créer un fichier de configuration par instance à sauvegarder. Le nom du fichier doit se terminer par `.conf` pour être pris en compte. Les fichiers doivent être regroupés dans un répertoire référencé par le paramètre `config-include-path`.

3.3.7 pgBackRest - Configuration PostgreSQL

- Adapter l'archivage dans le fichier `postgresql.conf`

```
archive_mode = on
wal_level = replica
archive_command = 'pgbackrest --stanza=erp_prod archive-push %p'
archive_timeout = '? min'   # à définir
```

Il est nécessaire d'activer l'archivage des journaux de transactions en positionnant le paramètre `archive_mode` à `on` et en définissant un niveau d'enregistrement d'informations dans les journaux de transactions (`wal_level`) supérieur ou égal à `replica` (ou `archive` avant la version 9.6).

pgBackRest fournit une commande permettant de simplifier la configuration de l'archivage. Pour l'utiliser, il faut configurer le paramètre `archive_command` pour qu'il utilise l'option `archive-push` de la commande `pgbackrest`. Il faut également fournir à cette commande le nom de la `stanza` à utiliser.

Comme pgBackRest n'archive que des journaux complets, il vaut mieux penser à mettre un `archive_timeout` adapté au RPO accepté. (S'il est nul, les auteurs recommandent plutôt un secondaire synchrone[9]).

3.3.8 pgBackRest - Configuration globale

- Fichier `pgbackrest.conf`
- Section **[global]** pour la configuration globale

```
[global]
process-max=1
repo1-path=/var/lib/pgbackrest
```

- **process-max** : nombre de processus maximum à utiliser pour la compression et le transfert des sauvegardes ;
- **repo1-path** : chemin où seront stockées les sauvegardes et les archives ;
- **repo-cipher-pass** : passphrase à utiliser pour chiffrer/déchiffrer le répertoire des sauvegardes ;
- **log-level-console** : par défaut à `warn`, définit le niveau de traces des commandes exécutées en console.

[9] https://github.com/pgbackrest/pgbackrest/issues/2233#issuecomment-1831406999

3.3.9 pgBackRest - Configuration de la rétention

- Type de rétention des sauvegardes complètes

```
repo1-retention-full-type=count|time
```

- **Nombre** de sauvegardes complètes

```
repo1-retention-full=2
```

- **Nombre** de sauvegardes différentielles

```
repo1-retention-diff=3
```

La politique de rétention des sauvegardes complètes peut être configurée avec l'option `repo1-retention-full-type`
Elle peut prendre deux valeurs :

- `count` : le nombre de sauvegardes à conserver, c'est la valeur par défaut ;
- `time` : un nombre de jours pendant lequel on doit pouvoir restaurer, c'est-à-dire que l'on doit avoir au moins une sauvegarde plus vieille que ce nombre de jours.

Voici un exemple pour illustrer le mode de rétention `time`, dont le fonctionnement n'est pas très intuitif. Si l'on dispose des trois sauvegardes complètes suivantes :

- F1 : 25 jours ;
- F2 : 20 jours ;
- F3 : 10 jours.

Avec une rétention de 15 jours, seule la sauvegarde F1 sera supprimée. F2 sera conservée, car il doit exister au moins une sauvegarde de plus de 15 jours pour garantir de pouvoir restaurer pendant cette période.

Il est possible de différencier le nombre de sauvegardes complètes et différentielles. La rétention pour les sauvegardes différentielles ne peut être définie qu'en nombre.

Lorsqu'une sauvegarde complète expire, toutes les sauvegardes différentielles et incrémentales qui lui sont associées expirent également.

3.3.10 pgBackRest - Configuration SSH

- Utilisateur `postgres` pour les serveurs PostgreSQL
- Échanger les clés SSH publiques entre les serveurs PostgreSQL et le serveur de sauvegarde
- Configurer `repo1-host*` dans la `pgbackrest.conf`

Dans le cadre de la mise en place de sauvegardes avec un stockage des données sur un serveur tiers, pgBackRest fonctionnera par SSH.

Il est donc impératif d'autoriser l'authentification SSH par clé, et d'échanger les clés publiques entre les différents serveurs hébergeant les instances PostgreSQL et le serveur de sauvegarde.

Il faudra ensuite adapter les paramètres `repo1-host*` dans la configuration de pgBackRest.

- **repo1-host** : hôte à joindre par SSH ;
- **repo1-host-user** : utilisateur pour la connexion SSH ;
- …

3.3.11 pgBackRest - Configuration TLS

- Alternative au SSH
- `{repo1|pg1}-host-type = tls`
- paramètres `tls-server-{address|auth|cert|key|ca}`
- paramètres `repo1-host-{cert|key|ca}`
- paramètres `pg1-host-{cert|key|ca}`
- `pgbackrest server`

Il existe une alternative à l'utilisation de SSH qui consiste à configurer un serveur TLS en valorisant le paramètre `repo1-host-type` et `pg1-host-type` à `tls` (défaut : `ssh`). La configuration du serveur se fait ensuite avec les paramètres :

- `tls-server-address` : adresse IP sur laquelle le serveur écoute pour servir des requêtes clients ;
- `tls-server-auth` : la liste des clients autorisés à se connecter sous la forme `<client-cn>=<stanza>` ;
- `tls-server-ca-file` : certificat de l'autorité ;

- `tls-server-cert-file` : certificat du serveur ;
- `tls-server-key-file` : clé du serveur.

Il faut ensuite configurer l'accès au dépôt de sauvegarde :

- `repo1-host-type=tls` : la connexion au dépôt utilise TLS ;
- `repo1-host-cert-file` : certificat pour se connecter au dépôt ;
- `repo1-host-key-file` : clé pour se connecter au dépôt ;
- `repo1-host-ca-file` : certificat de l'autorité.

Exemple de configuration :

```
[global]
repo1-host=backrest-srv
repo1-host-user=backrest
repo1-host-type=tls
repo1-host-cert-file=/etc/certs/srv1-cert.pem
repo1-host-key-file=/etc/certs/srv1-key.pem
repo1-host-ca-file=/etc/certs/CA-cert.pem

tls-server-address=*
tls-server-cert-file=/etc/certs/srv1-cert.pem
tls-server-key-file=/etc/certs/srv1-key.pem
tls-server-ca-file=/etc/certs/CA-cert.pem
tls-server-auth=backrest-srv=main

[main]
pg1-path=/var/lib/pgsql/14/data
```

Sur le serveur de sauvegarde, la configuration est similaire :

- `pg1-host-type=tls` : la connexion au serveur PostgreSQL utilise TLS ;
- `pg1-host-cert-file` : certificat pour se connecter au serveur de bases de données ;
- `pg1-host-key-file` : certificat pour se connecter au serveur de bases de données ;
- `pg1-host-ca-file` : certificat de l'autorité.

Exemple de configuration du serveur de sauvegarde :

```
[global]
repo1-path=/var/lib/pgbackrest
repo1-retention-full=2

tls-server-address=*
tls-server-cert-file=/etc/certs/backrest-srv-cert.pem
tls-server-key-file=/etc/certs/backrest-srv-key.pem
tls-server-ca-file=/etc/certs/CA-cert.pem
tls-server-auth=srv1=main

[main]
pg1-host=srv1
pg1-port=5432
pg1-path=/var/lib/pgsql/14/data
```

```
pg1-host-type=tls
pg1-host-cert-file=/etc/certs/backrest-srv-cert.pem
pg1-host-key-file=/etc/certs/backrest-srv-key.pem
pg1-host-ca-file=/etc/certs/CA-cert.pem
```

Le serveur TLS doit ensuite être démarré avec la commande `pgbackrest server`. Un service est prévu à cet effet et installé automatiquement sur les distributions de type RedHat et Debian.

Un ping vers le serveur TLS peut être testé avec la commande `pgbackrest server-ping <hote>`. Suivant les distributions, il peut être nécessaire d'ouvrir le port 8432 (valeur par défaut de `tls-server-port`).

```
[postgres@backrest log]$ pgbackrest server-ping srv1
INFO: server-ping command begin 2.41: [srv1] --exec-id=7467-76e4b8cf
 --log-level-console=info --tls-server-address=*
INFO: server-ping command end: completed successfully (47ms)
```

Génération des clés et certificats auto-signés :

```
# Générer une clé privée et un certificat pour l'autorité de certification
openssl req -new -x509 \
                 -days 365 \
         -nodes \
         -out CA-cert.pem \
         -keyout CA-key.pem \
         -subj "/CN=root-ca"

# Générer une clé privée et demande de certificat (CSR)
openssl req -new -nodes \
                 -out backrest-srv-csr.pem \
         -keyout backrest-srv-key.pem \
         -subj "/CN=backrest-srv"
openssl req -new -nodes \
                 -out srv1-csr.pem \
         -keyout srv1-key.pem \
         -subj "/CN=srv1"

# Générer le certificat signé
openssl x509 -req -in backrest-srv-csr.pem \
                  -days 365 \
          -CA CA-cert.pem \
          -CAkey CA-key.pem \
          -CAcreateserial \
          -out backrest-srv-crt.pem
openssl x509 -req -in srv1.csr
                  -days 365 \
          -CA CA-cert.pem \
          -CAkey CA-key.pem \
          -CAcreateserial \
          -out srv1-crt.pem
```

3.3.12 pgBackRest - Configuration par instance

- Une section par instance
  - appelée `stanza`

Après avoir vu les options globales, nous allons voir à présent les options spécifiques à chaque instance à sauvegarder.

3.3.13 pgBackRest - Exemple configuration par instance

- Section spécifique par instance
- Permet d'adapter la configuration aux différentes instances
- Exemple

```
[erp_prod]
pg1-path=/var/lib/pgsql/10/data
```

Une `stanza` définit l'ensemble des configurations de sauvegardes pour un cluster PostgreSQL spécifique. Chaque section `stanza` définit l'emplacement du répertoire de données ainsi que l'hôte/utilisateur si le cluster est distant. Chaque configuration de la partie globale peut être surchargée par `stanza`.

Le nom de la `stanza` est important et doit être significatif car il sera utilisé lors des tâches d'exploitation pour identifier l'instance cible.

Il est également possible d'ajouter ici des `recovery-option` afin de personnaliser les options du `postgresql.auto.conf` qui sera généré automatiquement à la restauration d'une sauvegarde.

3.3.14 pgBackRest - Initialiser le répertoire de stockage des sauvegardes

- Pour initialiser le répertoire de stockage des sauvegardes

```
$ sudo -u postgres pgbackrest --stanza=erp_prod stanza-create
```

- Vérifier la configuration de l'archivage

```
$ sudo -u postgres pgbackrest --stanza=erp_prod check
```

La commande d'initialisation doit être lancée sur le serveur où se situe le répertoire de stockage après que la `stanza` ait été configurée dans `pgbackrest.conf`.

La commande `check` valide que pgBackRest et le paramètre `archive_command` soient correctement configurés. Les commandes `pg_create_restore_point('pgBackRest Archive Check')` et `pg_switch_wal()` sont appelées à cet effet pour forcer PostgreSQL à archiver un segment WAL.

3.3.15 pgBackRest - Effectuer une sauvegarde

- Pour déclencher une nouvelle sauvegarde complète

```
$ sudo -u postgres pgbackrest --stanza=erp_prod --type=full backup
```

- Types supportés : `incr`, `diff`, `full`
- La plupart des paramètres peuvent être surchargés

La sauvegarde accepte de nombreux paramètres dont :

- `--archive-copy` : archive les WAL dans la sauvegarde en plus de les mettre dans le dépôt de WAL ;
- `--backup-standby` : déclenche la sauvegarde sur un serveur secondaire ;
- `--no-online` : fait une sauvegarde à froid ;
- `--resume` : reprend une sauvegarde précédemment échouée en conservant les fichiers qui n'ont pas changés ;
- `--start-fast` : exécuter le checkpoint immédiatement.

Exemple de sortie d'une sauvegarde complète :

```
$ sudo -u postgres pgbackrest --stanza=erp_prod --type=full backup |grep P00
P00   INFO: backup command begin 2.19: --log-level-console=info
--no-log-timestamp --pg1-path=/var/lib/pgsql/12/data --process-max=1
--repo1-path=/var/lib/pgsql/12/backups --repo1-retention-full=1
--stanza=erp_prod --type=full
P00   INFO: execute non-exclusive pg_start_backup() with label
"pgBackRest backup started at 2019-11-26 12:39:26":
backup begins after the next regular checkpoint completes
P00   INFO: backup start archive = 000000010000000000000005, lsn = 0/5000028
P00   INFO: full backup size = 24.2MB
P00   INFO: execute non-exclusive pg_stop_backup() and wait for all WAL
segments to archive
P00   INFO: backup stop archive = 000000010000000000000005, lsn = 0/5000100
P00   INFO: new backup label = 20191126-123926F
P00   INFO: backup command end: completed successfully
P00   INFO: expire command begin 2.19: --log-level-console=info
--no-log-timestamp --pg1-path=/var/lib/pgsql/12/data --process-max=1
--repo1-path=/var/lib/pgsql/12/backups --repo1-retention-full=1
--stanza=erp_prod --type=full
P00   INFO: expire full backup 20191126-123848F
P00   INFO: remove expired backup 20191126-123848F
P00   INFO: expire command end: completed successfully
```

La commande se charge automatiquement de supprimer les sauvegardes devenues obsolètes.

Il est possible d'ajouter des annotations aux sauvegardes comme ceci :

```
$ sudo -u postgres pgbackrest
  --stanza=erp_prod
  --type=full
  --annotation=desc="Premier backup"
  backup
```

L'annotation peut être observé en affichant les informations du *backup set*.

3.3.16 pgBackRest - Lister les sauvegardes

- Lister les sauvegardes présentes et leur taille

```
$ sudo -u postgres pgbackrest --stanza=erp_prod info
```

- ou une sauvegarde spécifique (*backup set*)

```
$ sudo -u postgres pgbackrest --stanza=erp_prod --set 20221026-071751F
 ↪  info
```

Exemple de sortie des commandes :

```
$ sudo -u postgres pgbackrest --stanza=erp_prod info
stanza: erp_prod
```

```
    status: ok
    cipher: none

    db (current)
        wal archive min/max (14): 000000030000000000000019/000000030000000000000001B

        full backup: 20221026-071751F
            timestamp start/stop: 2022-10-26 07:17:51 / 2022-10-26 07:17:57
            wal start/stop: 000000030000000000000001B / 000000030000000000000001B
            database size: 25.2MB, database backup size: 25.2MB
            repo1: backup set size: 3.2MB, backup size: 3.2MB

$ sudo -u postgres pgbackrest --stanza=erp_prod --set 20221026-071751F info
stanza: erp_prod
    status: ok
    cipher: none

    db (current)
        wal archive min/max (14): 000000030000000000000019/000000030000000000000001B

        full backup: 20221026-071751F
            timestamp start/stop: 2022-10-26 07:17:51 / 2022-10-26 07:17:57
            wal start/stop: 000000030000000000000001B / 000000030000000000000001B
            lsn start/stop: 0/1B000028 / 0/1B000100
            database size: 25.2MB, database backup size: 25.2MB
            repo1: backup set size: 3.2MB, backup size: 3.2MB
            database list: postgres (13748)
            annotation(s)
                desc: Premier backup
```

3.3.17 pgBackRest - Planification

- Pas de planificateur intégré
  - le plus simple est d'utiliser `cron`

La planification des sauvegardes peut être faite par n'importe quel outil de planification de tâches, le plus connu étant `cron`.

pgBackRest maintient les traces de ses activités par défaut dans `/var/log/pgbackrest` avec un niveau de traces plus élevé qu'en console. Il n'est donc généralement pas nécessaire de gérer cela au niveau de la planification.

3.3.18 pgBackRest - Dépôts

- Plusieurs dépôts simultanés possibles
  - Sauvegarde des journaux en parallèle
  - `--repo1-option=…` , appel avec `--repo=1`
- Types : POSIX (NFS, ssh), CIFS, SFTP
- Cloud : S3, Azure, GFS

pgBackRest permet de maintenir plusieurs dépôts de sauvegarde simultanément[10].

Un intérêt est de gérer des rétentions différentes. Par exemple un dépôt local contiendra juste les dernières sauvegardes et journaux, alors qu'un deuxième dépôt sera sur un autre site plus lointain, éventuellement moins cher, et/ou une rétention supérieure.

Les propriétés des différents dépôts (type, chemin, rétention…) se définissent avec les options `repo1-path`, `repo2-path`, etc. Désigner un dépôt particulier se fait avec `--repo=1` par exemple.

Une sauvegarde se fait en désignant le dépôt, mais l'archivage est simultané sur tous les dépôts. L'archivage asynchrone est conseillé dans ce cas.

Les types de dépôts supportés sont ceux montés sur le serveur ou accessibles par ssh, NFS (avec la même attention aux options de montage que pour PostgreSQL[11]), CIFS (avec des restrictions sur les liens symboliques ou le fsync), mais aussi ceux à base de *buckets* : S3 ou compatible, Google Cloud, et Azure Blob.

[10] https://pgbackrest.org/configuration.html#section-repository
[11] https://docs.postgresql.fr/current/creating-cluster.html#CREATING-CLUSTER-FILESYSTEM

3.3.19 pgBackRest - bundling et sauvegarde incrémentale en mode block

- Regrouper les petits fichiers dans des bundles

```
repo1-bundle=y
```

- Sauvegarde incrémentale en mode block (requiert le bundling)

```
repo1-bundle=y
repo1-block=y
```

« Bundling » des petits fichiers

Si une instances contient de nombreux petits fichiers (base aux nombreuses toutes petites tables, `pg_commit_ts` rempli à cause de `track_commit_timestamp` à `on`, très nombreuses petites partitions, chacune avec des fichiers annexes…), il est possible de les regrouper par paquets.

```
repo1-bundle=y
# défauts
repo1-bundle-limit=2MiB
repo-bundle-size=20MiB
```

Les bundles ne sont pas conservés en cas de backup interrompu puis redémarré. Les fichiers doivent être re-sauvegardés lors de la relance. Bundles et hard-links ne peuvent pas être utilisés ensemble.

Cette fonctionnalité est particulièrement utile avec un stockage comme S3 où le coût de création de fichier est prohibitif.

Sauvegarde incrémentale en mode bloc (2.46)

La sauvegarde incrémentale par bloc permet plus de granularité en divisant les fichiers en blocs qui peuvent être sauvegardés indépendamment. C'est particulièrement intéressant pour des fichiers avec peu de modifications, car pgBackRest ne sauvegardera que quelques blocs au lieu du fichier complet (les tables et index sont segmentés en fichiers de 1 Go). Cela permet donc d'économiser de l'espace dans le dépôt de sauvegarde et accélère les restaurations par delta.

La sauvegarde incrémentale par bloc doit être activée sur tous les types de sauvegardes : full, incrémentielle ou différentielle. Cela aura pour impact de rendre la sauvegarde full un peu plus grosse du fait de la création de fichier de cartographie des blocs. En revanche, les sauvegardes différentielles et incrémentielles suivantes pourront utiliser cette cartographie pour économiser de l'espace.

La taille du bloc pour un fichier donné est définie en fonction de l'âge et de la taille du fichier. Généralement, les fichiers les plus gros et/ou les plus anciens auront des tailles de bloc supérieures. Si un fichier est assez vieux, aucune cartographie ne sera crée.

Cette fonctionnalité nécessite le *bundling* et s'active ainsi :

```
repo1-block=y
repo1-bundle=y
```

3.3.20 pgBackRest - Restauration

- Effectuer une restauration

```
$ sudo -u postgres pgbackrest --stanza=erp_prod restore
```

- Nombreuses options à la restauration, notamment :
  - `--delta`
  - `--target` / `--type`

Exemple de sortie de la commande :

```
$ sudo -u postgres pgbackrest --stanza=erp_prod restore |grep P00

P00   INFO: restore command begin 2.19: --log-level-console=info
--no-log-timestamp --pg1-path=/var/lib/pgsql/12/data
--process-max=1 --repo1-path=/var/lib/pgsql/12/backups --stanza=erp_prod
P00   INFO: restore backup set 20191126-123926F
P00   INFO: write updated /var/lib/pgsql/12/data/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
```

L'option `--delta` permet de ne restaurer que les fichiers qui seraient différents entre la sauvegarde et le répertoire de données déjà présent sur le serveur. Elle permet de gagner beaucoup de temps pour reprendre une restauration qui a été interrompue pour une raison ou une autre, pour resynchroniser une instance qui a « décroché », pour restaurer une version légèrement antérieure ou postérieure dans du PITR.

La cible à restaurer peut être spécifiée avec `--target`, associé à `--type`. Par exemple, pour restaurer à une date précise sur une timeline précise :

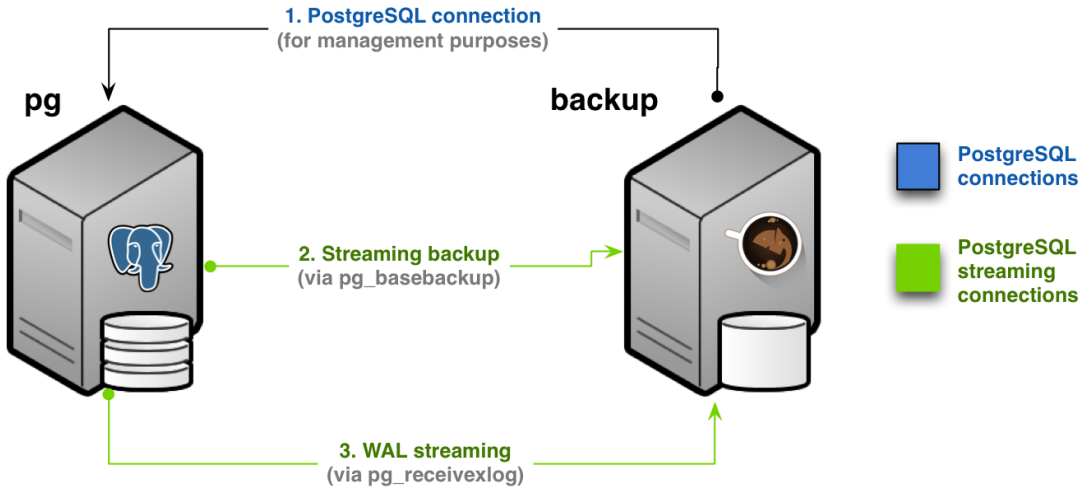
```
pgbackrest --stanza=instance --delta \
  --type=time --target='2020-07-16 11:07:00' \
  --target-timeline=4 \
  --target-action=pause \
  --set=20200716-102845F \
  restore
```

3.4 BARMAN - PRÉSENTATION GÉNÉRALE

- 2ndQuadrant Italia
- Langage: **python** >= 3.4
- OS: **Unix/Linux**
- Versions compatibles: **>= 8.3**
- License: **GPL3** (libre)
- Type d'interface: **CLI** (ligne de commande)

`barman` est un outil développé avec le langage python, compatible uniquement avec les environnements Linux/Unix. Il a été développé par la société 2ndQuadrant Italia (à présent partie de EDB) et distribué sous license GPL3.

3.4.1 Barman - Scénario « streaming-only »



Le scénario évoqué ci-dessus est communément appelé `streaming-only` puisqu'il ne requiert pas de connexion SSH pour les opérations de sauvegardes et d'archivage. Il faudra quand même configurer le SSH pour rendre possible la restauration depuis un serveur dédié ou faciliter la restauration en local.

En effet, les outils `pg_basebackup` et `pg_receivewal` sont utilisés pour ces opérations et se basent donc uniquement sur le protocole de réplication. Cela a pour avantage que les améliorations faites aux outils dans le cadre des mises à jour majeures de PostgreSQL sont disponible directement dans Barman.

Par exemple :

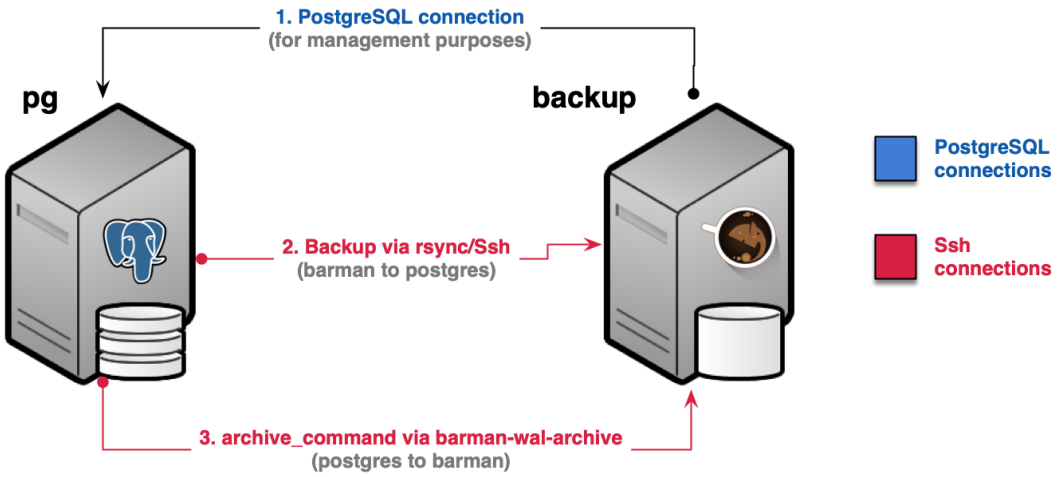
- la possibilité d'utiliser `pg_stat_progress_basebackup` pour la supervision ;
- les fichiers manifestes de sauvegarde et la vérification des sauvegardes ;
- la compression des sauvegardes.

Afin de garantir que l'instance sauvegardée conserve bien les WAL nécessaires, Barman permet de créer automatiquement un slot de réplication. Il se chargera également de démarrer `pg_receivewal` grâce à sa tâche de maintenance programmée en crontab.

L'archivage peut être configuré à la place ou en plus du streaming des WAL.

Ce mode de sauvegarde permet de sauvegarder un serveur PostgreSQL installé sous Windows.

3.4.2 Barman - Scénario « rsync-over-ssh »



Ce deuxième scénario se base donc sur une connexion SSH afin de réaliser les sauvegardes et récupérer les archives des journaux WAL.

Cette méthode ne permet pas de compresser les sauvegardes mais permet de faire de la déduplication avec des *hard links* et de bénéficier de la parallélisation.

3.4.3 Barman - Sauvegardes

- Type de sauvegarde : **physique/PITR** (à chaud)
- Type de stockage : **local** ou **pull**
- Planification : **crontab**
- Méthodes :
  - **pg_backup_start() / rsync / pg_backup_stop()**
  - **pg_basebackup / pg_receivewal**
- Incrémentales : si **rsync + hardlink**
- Compression des WAL

Barman gère uniquement des sauvegardes physiques.

Il peut fonctionner soit en local (directement sur le serveur hébergeant l'instance à sauvegarder) pour un stockage local des sauvegardes, et peut aussi être exécuté depuis un serveur distant, déléguant ainsi l'ordonnancement, la compression et le stockage des données.

La technique utilisée pour la prise de sauvegarde repose sur le mécanisme interne standard et historique : `pg_backup_start()`, copie des fichiers, `pg_backup_stop()`.

Contrairement aux autres outils présentés, Barman peut également se servir de `pg_basebackup` et `pg_receivewal` pour récupérer les sauvegardes et les archives des journaux WAL.

Il est possible d'activer la dé-duplication de fichiers entre deux sauvegardes lorsque la méthode via `rsync` est employée.

3.4.4 Barman - Sauvegardes (suite)

- Limitation du débit réseau lors des transferts
- Compression des données lors des transferts via le réseau
- Sauvegardes concurrentes
- Hook pre/post sauvegarde
- Hook pre/post archivage WAL
- Compression WAL : `gzip`, `bzip2`, `pigz`, `pbzip2`, etc.
- Compression des données via `pg_basebackup`

Barman supporte la limitation du débit réseau lors du transfert des données sur un serveur tiers, ainsi que la compression des données à la volée le temps du transfert.

Quatre niveaux de scripts ancrés (*hooks*) sont possibles :

- **avant la sauvegarde** ;
- **après la sauvegarde** ;
- **avant l'archivage d'un WAL** ;
- **après l'archivage d'un WAL**.

Attention, l'opération d'archivage citée ici est celle effectuée par Barman lorsqu'il déplace et compresse un WAL à partir du répertoire `incoming_wals/` vers le répertoire `wals/`, il ne s'agit pas de l'archivage au sens PostgreSQL.

3.4.5 Barman - Politique de rétention

- **Durée** (jour/semaine)
- **Nombre** de sauvegardes

La politique de rétention peut être exprimée soit en nombre de sauvegardes à conserver, soit en fenêtre de restauration : une semaine, deux mois, etc.

3.4.6 Barman - Restauration

- Locale ou à distance
- Point dans le temps : date, identifiant de transaction, timeline ou point de restauration

La restauration d'une sauvegarde peut se faire soit localement, si les sauvegardes sont stockées en local, soit à distance. Dans ce dernier cas, les données à restaurer seront transférées via SSH.

Plusieurs types de point dans le temps peuvent être utilisés comme cible :

- la date ;
- un identifiant de transaction ;
- une timeline (en cas de divergence de timeline, `barman` peut restaurer les transactions issues d'une timeline précise) ;

- un point de restauration créé par un appel préalable à la fonction :
  - `pg_create_restore_point()`.

3.4.7 Barman - Installation

- Accéder au dépôt communautaire PGDG
- Installer les paquets `barman` et `barman-cli`

Barman est disponible sur le dépôt communautaire maintenu par la communauté PostgreSQL pour les systèmes d'exploitation disposant des gestionnaires de paquet au format deb (Debian, Ubuntu…)[12] ou rpm (Red Hat, Rocky Linux, CentOS, Fedora…)[13].

Il est recommandé de manière générale de privilégier une installation à partir des paquets issus du PGDG plutôt que par les sources, essentiellement pour des raisons de maintenance.

[12] https://apt.postgresql.org/pub/repos/apt/
[13] https://yum.postgresql.org/

3.4.8 Barman - Utilisation

```
usage: barman [-h] [-v] [-c CONFIG] [--color {never,always,auto}] [-q]
 ↪  [-d]
              [-f {json,console}]

          {archive-wal,backup,check,check-backup,check-wal-archive,cron,

 ↪  delete,diagnose,generate-manifest,get-wal,help,keep,list-backup,
          list-backups,list-files,list-server,list-servers,put-wal,

 ↪  rebuild-xlogdb,receive-wal,recover,replication-status,show-backup,

 ↪  show-backups,show-server,show-servers,status,switch-wal,switch-xlog,
          sync-backup,sync-info,sync-wals,verify,verify-backup}

[...]
optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -c CONFIG, --config CONFIG
                      uses a configuration file (defaults: ~/.barman.conf,
                        /etc/barman.conf, /etc/barman/barman.conf)
  --color {never,always,auto}, --colour {never,always,auto}
                        Whether to use colors in the output (default:
 ↪  'auto')
  -q, --quiet           be quiet (default: False)
  -d, --debug           debug output (default: False)
  -f {json,console}, --format {json,console}
                        output format (default: 'console')
```

Barman propose différentes commandes pouvant être passées en argument afin de contrôler les actions.

L'usage de ces différentes commandes sera détaillé ultérieurement.

L'option `-c` (ou `--config`) permet d'indiquer l'emplacement du fichier de configuration. L'option `-q` (ou `--quiet`) désactive l'envoi de messages sur la sortie standard.

3.4.9 Barman - Configuration

- `/etc/barman.conf`
- Format `INI`
- Configuration générale dans la section `[barman]`
- Chaque instance à sauvegarder doit avoir sa propre section
- Un fichier de configuration par instance via la directive :

```
configuration_files_directory = /etc/barman.d
```

Le format de configuration `INI` permet de définir des sections, qui sont matérialisées sous la forme d'une ligne : `[nomdesection]`.

Barman s'attend à lire un fichier de configuration contenant la section `[barman]`, contenant les paramètres de configuration globaux, et une section par instance à sauvegarder, le nom de la section définissant ainsi le nom de l'instance.

Pour des questions de lisibilité, il est possible de créer un fichier de configuration par instance à sauvegarder. Ce fichier doit alors se trouver (par défaut) dans le dossier `/etc/barman.d`. Le nom du fichier doit se terminer par `.conf` pour être pris en compte.

3.4.10 Barman - Configuration utilisateur

- Utilisateur système `barman`

L'utilisateur système `barman` est utilisé pour les connexions SSH. Il faut donc penser à générer ses clés RSA, les échanger et établir une première connexion avec les serveurs hébergeant les instances PostgreSQL à sauvegarder.

3.4.11 Barman - Configuration SSH

- Utilisateur `postgres` pour les serveurs PostgreSQL
- Utilisateur `barman` pour le serveur de sauvegardes
- Générer les clés SSH (RSA) des utilisateurs système `postgres` (serveurs PG) et `barman` (serveur barman)
- Échanger les clés SSH publiques entre les serveurs PostgreSQL et le serveur de sauvegarde
- Établir manuellement une première connexion SSH entre chaque machine
- Inutile si utilisation de `pg_basebackup` / `pg_receivewal`

Dans le cadre de la mise en place de sauvegardes avec un stockage des données sur un serveur tiers, la plupart des outils et méthodes historiques de sauvegardes reposent sur le protocole SSH et des outils tels que `rsync` pour assurer les transferts au travers du réseau.

Afin d'automatiser ces transferts via le protocole SSH, il est impératif d'autoriser l'authentification SSH par clé, et d'échanger les clés publiques entre les différents serveurs hébergeant les instances PostgreSQL et le serveur de sauvegarde.

3.4.12 Barman - Configuration PostgreSQL

- Adapter la configuration de l'archivage dans le fichier `postgresql.conf` :

```
wal_level = 'replica'
archive_mode = on
archive_command = 'barman-wal-archive backup-srv pgsrv %p'
```

- … ou paramétrer la réplication si utilisation de `pg_basebackup` / `pg_receivewal`

Le paramétrage de l'archivage des journaux de transactions reste classique. La directive `archive_command` doit faire appel directement à l'outil système en charge du transfert du fichier.

Le paramètre `archive_mode` peut prendre la valeur `always` pour permettre un archivage à partir des serveurs secondaires.

Depuis la version 2.6 de Barman, il est recommandé d'utiliser la commande `barman-wal-archive` intégrée (fournie par le paquet `barman-cli`) pour gérer l'archivage. Cette commande interagit directement avec Barman pour recevoir le fichier, écrire son contenu *via* `fsync` et l'envoyer dans le répertoire *incomming* adapté. Cela réduit donc le risque de corruption, perte de données ou simplement d'erreur de répertoire.

3.4.13 Barman - Configuration globale

- `barman.conf`

```
[barman]
barman_home = /var/lib/barman
barman_user = barman
log_file = /var/log/barman/barman.log
log_level = INFO
configuration_files_directory = /etc/barman.d
```

- **barman_home** : répertoire racine de travail de Barman, contenant les sauvegardes et les journaux de transactions archivés ;
- **barman_user** : utilisateur système ;
- **log_file** : fichier contenant les traces Barman ;
- **configuration_files_directory**: chemin vers le dossier d'inclusion des fichiers de configuration supplémentaires (défaut : `/etc/barman.d`) ;
- **log_level** : niveau de verbosité des traces, par défaut `INFO`.

3.4.14 Barman - Configuration sauvegardes

- Configuration globale des options de sauvegarde

```
compression = gzip
backup_compression = gzip
immediate_checkpoint = false
basebackup_retry_times = 0
basebackup_retry_sleep = 30
```

- **compression** : méthode de compression des journaux de transaction - sont disponibles : `gzip`, `bzip2`, `custom`, laissant la possibilité d'utiliser l'utilitaire de compression de son choix (dé-

faut : `gzip`) ;
- **backup_compression** : méthode utilisée par Barman pour compresser la sauvegarde, les options disponibles dépendent de la version de PostgreSQL utilisée (la version 15 apporte beaucoup de nouveautés à ce niveau) ;
- **immediate_checkpoint** : force la création immédiate d'un checkpoint impliquant une augmentation des écritures, le but étant de débuter la sauvegarde le plus rapidement possible (defaut : `off`) ;
- **basebackup_retry_times** : nombre de tentative d'écriture d'un fichier - utile pour relancer la copie d'un fichier en cas d'échec sans compromettre le déroulement global de la sauvegarde ;
- **basebackup_retry_sleep** : spécifié en secondes, il s'agit ici de l'intervalle de temps entre deux tentatives de copie d'un fichier en cas d'échec.

3.4.15 Barman - Configuration réseau

- Possibilité de réduire la bande passante
- Et de compresser le trafic réseau
- Exemple

```
bandwidth_limit = 4000
network_compression = false
```

- **bandwidth_limit** : limitation de l'utilisation de la bande passante réseau lors du transfert de la sauvegarde, s'exprime en `kbps` (par défaut à `0`, autrement dit pas de limitation) ;
- **network_compression** : activation de la compression à la volée des données lors du transfert réseau de la sauvegarde - utilisé à la sauvegarde ou lors d'une restauration (défaut : `false`).

3.4.16 Barman - Configuration rétention

- Configuration de la rétention en nombre de sauvegardes
- Ou en « fenêtre de restauration », en jours, semaines ou mois
- Déclenchement d'une erreur en cas de sauvegarde trop ancienne
- Exemple

```
minimum_redundancy = 5
retention_policy = RECOVERY WINDOW OF 7 DAYS
last_backup_maximum_age = 2 DAYS
```

- **minimum_redundancy** : nombre minimum de sauvegardes à conserver - si ce n'est pas respecté, Barman empêchera la suppression (défaut : `0`) ;
- **retention_policy** : définit la politique de rétention en s'exprimant soit en nombre de sauvegarde via la syntaxe `REDUNDANCY <valeur>`, soit en fenêtre de restauration via la syntaxe `RECOVERY OF <valeur> {DAYS | WEEKS | MONTHS}` (défaut : aucune rétention appliquée) ;
- **last_backup_maximum_age** : expression sous la forme `<value> {DAYS | WEEKS | MONTHS}`, définit l'âge maximal de la dernière sauvegarde - si celui-ci n'est pas respecté, lors de l'utilisation de la commande `barman check`, une erreur sera levée.

3.4.17 Barman - Configuration des hooks

- Lancer des scripts avant ou après les sauvegardes
- Et avant ou après le traitement du WAL archivé par Barman
- Exemple :

```
pre_backup_script = ...
post_backup_script = ...
pre_archive_script = ...
post_archive_script = ...
```

Barman offre la possibilité d'exécuter des commandes externes (scripts) avant et/ou après les opérations de sauvegarde et les opérations d'archivage des journaux de transaction.

Attention, la notion d'archivage de journal de transactions dans ce contexte ne concerne pas l'archivage réalisé depuis l'instance PostgreSQL, qui copie les WAL dans un répertoire `<incoming>` sur le serveur Barman, mais bien l'opération de récupération du WAL depuis ce répertoire `<incoming>`.

3.4.18 Barman - Configuration d'un dépôt synchronisé

- Copie à l'identique du dépôt d'origine
- Sur le dépôt à synchroniser :
  - `primary_ssh_command`
- Commandes :
  - `barman sync-info --primary <instance> <ID-sauvegarde>`
  - `barman sync-backup <instance> <ID-sauvegarde>`
  - `barman sync-wal <instance>`

Barman permet de créer une copie d'un dépôt barman pour répondre à des besoins de redondance géographique. Il suffit pour cela de valoriser le paramètre `primary_ssh_command` pour que le serveur barman client se connecte au serveur principal et duplique les sauvegardes et WAL.

La commande `barman sync-info --primary <instance> <ID-sauvegarde>` permet d'afficher les informations de synchronisation. Le processus de copie est lancé automatiquement par la tâche de maintenance automatisée. Il est aussi possible de lancer la synchronisation manuellement pour une sauvegarde en particulier avec `barman sync-backup <instance> <ID-sauvegarde>` ou pour les WAL avec `barman sync-wal <instance>`.

3.4.19 Barman - Configuration par instance

- `configuration_files_directory`
  - un fichier de configuration par instance
- Ou une section par instance

Après avoir vu les options globales, nous allons voir à présent les options spécifiques à chaque instance à sauvegarder.

Afin de conserver une certaine souplesse dans la gestion de la configuration Barman, il est recommandé de paramétrer la directive `configuration_files_directory` de la section `[barman]` afin de pouvoir charger d'autres fichiers de configuration, permettant ainsi d'isoler la section spécifique à chaque instance à sauvegarder dans son propre fichier de configuration.

3.4.20 Barman - Exemple configuration par instance

- Section spécifique par instance
- Permet d'adapter la configuration aux différentes instances
- Exemple

```
[pgsrv]
description = "PostgreSQL Instance pgsrv"
ssh_command = ssh postgres@pgsrv
conninfo = host=pgsrv user=postgres dbname=postgres
backup_method = rsync
reuse_backup = link
backup_options = exclusive_backup
archiver = on
```

La première ligne définit le nom de la section. Ce nom est important et doit être significatif car il sera utilisé lors des tâches d'exploitation pour identifier l'instance cible.

L'idéal est d'utiliser le nom d'hôte ou l'adresse IP du serveur si celui-ci n'héberge qu'une seule instance.

- **description** : chaîne de caractère servant de descriptif de l'instance ;
- **ssh_command** : commande shell utilisée pour établir la connexion `ssh` vers le serveur hébergeant l'instance à sauvegarder ;
- **conninfo** : chaîne de connexion PostgreSQL.

Tous les autres paramètres, à l'exception de `log_file` et `log_level`, peuvent être redéfinis pour chaque instance.

3.4.21 Barman - Exemple configuration Streaming Only

```
[pgsrv]
description =  "Sauvegarde de pgsrv via Streaming Replication"
conninfo = host=pgsrv user=barman dbname=postgres
streaming_conninfo = host=pgsrv user=streaming_barman
backup_method = postgres
streaming_archiver = on
create_slot = auto
slot_name = barman
```

- `barman replication-status pgsrv`

La commande `barman replication-status` permet d'afficher l'état de la réplication :

```
$ barman replication-status pgsrv
Status of streaming clients for server 'pgsrv':
  Current LSN on master: 0/140001B0
  Number of streaming clients: 1

  1. Async WAL streamer
     Application name: barman_receive_wal
     Sync stage      : 3/3 Remote write
     Communication   : Unix domain socket
     User name       : barman
     Current state   : streaming (async)
     Replication slot: barman
     WAL sender PID  : 29439
     Started at      : 2022-10-17 14:54:02.122742+00:00
     Sent LSN   : 0/140001B0 (diff: 0 B)
     Write LSN  : 0/140001B0 (diff: 0 B)
     Flush LSN  : 0/14000000 (diff: -432 B)
```

3.4.22 Barman - Vérification de la configuration

- La commande `show-server` montre la configuration

```
$ sudo -u barman barman show-server {<instance> | all}
```

- La commande `check` effectue des tests pour la valider

```
$ sudo -u barman barman check {<instance> | all}
$ sudo -u barman barman check {<instance> | all} --nagios
```

La commande `show-server` permet de visualiser la configuration de Barman pour l'instance spécifiée, ou pour toutes les instances si le mot-clé `all` est utilisé.

La commande `check` vérifie le bon paramétrage de Barman pour l'instance spécifiée, ou pour toutes les instances si le mot-clé `all` est utilisé.

Elle permet de s'assurer que les points clés sont fonctionnels, tels que l'accès SSH, l'archivage des journaux de transaction (`archive_command`, `archive_mode`…), la politique de rétention, la compression, etc.

Il est possible d'utiliser l'option `--nagios` qui permet de formater la sortie de la commande `check` et de l'utiliser en tant que sonde Nagios.

Exemple de sortie de la commande `show-server` :

```
$ barman show-server pgsrv
Server pgsrv:
  active: True
  archive_command: None
  archive_mode: None
  archiver: True
  archiver_batch_size: 0
  backup_directory: /var/lib/barman/pgsrv
  backup_method: rsync
  backup_options: BackupOptions(['exclusive_backup'])
  bandwidth_limit: None
  barman_home: /var/lib/barman
  barman_lock_directory: /var/lib/barman
  basebackup_retry_sleep: 30
  basebackup_retry_times: 0
  basebackups_directory: /var/lib/barman/pgsrv/base
  check_timeout: 30
  compression: None
  conninfo: host=pgsrv user=postgres dbname=postgres
  create_slot: manual
  current_xlog: None
  custom_compression_filter: None
  custom_decompression_filter: None
  data_directory: None
  description: PostgreSQL Instance pgsrv
  disabled: False
  errors_directory: /var/lib/barman/pgsrv/errors
  immediate_checkpoint: False
  incoming_wals_directory: /var/lib/barman/pgsrv/incoming
  is_in_recovery: None
  is_superuser: None
  last_backup_maximum_age: None
  max_incoming_wals_queue: None
  minimum_redundancy: 0
  msg_list: []
  name: pgsrv
  network_compression: False
  parallel_jobs: 1
  passive_node: False
  path_prefix: None
  pgespresso_installed: None
  post_archive_retry_script: None
  post_archive_script: None
  post_backup_retry_script: None
  post_backup_script: None
  post_delete_retry_script: None
  post_delete_script: None
  post_recovery_retry_script: None
  post_recovery_script: None
  post_wal_delete_retry_script: None
  post_wal_delete_script: None
  postgres_systemid: None
  pre_archive_retry_script: None
  pre_archive_script: None
  pre_backup_retry_script: None
  pre_backup_script: None
```

```
  pre_delete_retry_script: None
  pre_delete_script: None
  pre_recovery_retry_script: None
  pre_recovery_script: None
  pre_wal_delete_retry_script: None
  pre_wal_delete_script: None
  primary_ssh_command: None
  recovery_options: RecoveryOptions([])
  replication_slot: None
  replication_slot_support: None
  retention_policy: None
  retention_policy_mode: auto
  reuse_backup: link
  server_txt_version: None
  slot_name: None
  ssh_command: ssh postgres@pgsrv
  streaming_archiver: False
  streaming_archiver_batch_size: 0
  streaming_archiver_name: barman_receive_wal
  streaming_backup_name: barman_streaming_backup
  streaming_conninfo: host=pgsrv user=postgres dbname=postgres
  streaming_wals_directory: /var/lib/barman/pgsrv/streaming
  synchronous_standby_names: None
  tablespace_bandwidth_limit: None
  wal_retention_policy: main
  wals_directory: /var/lib/barman/pgsrv/wals
```

Exemple de sortie de la commande `check` :

```
$ barman check pgsrv
Server pgsrv:
  PostgreSQL: OK
  superuser or standard user with backup privileges: OK
  PostgreSQL streaming: OK
  wal_level: OK
  replication slot: OK
  directories: OK
  retention policy settings: OK
  backup maximum age: OK (no last_backup_maximum_age provided)
  backup minimum size: OK (33.6 MiB)
  wal maximum age: OK (no last_wal_maximum_age provided)
  wal size: OK (0 B)
  compression settings: OK
  failed backups: OK (there are 0 failed backups)
  minimum redundancy requirements: OK (have 2 backups, expected at least 0)
  pg_basebackup: OK
  pg_basebackup compatible: OK
  pg_basebackup supports tablespaces mapping: OK
  systemid coherence: OK
  pg_receivexlog: OK
  pg_receivexlog compatible: OK
  receive-wal running: OK
  archiver errors: OK
```

3.4.23 Barman - Statut

- La commande `status` affiche des informations détaillées
  - sur la configuration Barman
  - sur l'instance spécifiée
- Exemple

```
$ sudo -u barman barman status {<instance> | all}
```

La commande `status` retourne de manière détaillée le statut de l'instance spécifiée, ou de toutes si le mot-clé `all` est utilisé.

Les informations renvoyées sont, entre autres :

- la description extraite du fichier de configuration de Barman ;
- la version de PostgreSQL ;
- si l'extension `pgespresso` est utilisée ;
- l'emplacement des données sur l'instance (`PGDATA`) ;
- la valeur de l'`archive_command` ;
- des informations sur les journaux de transactions :
  - position courante
  - dernier segment archivé
- des informations sur les sauvegardes :
  - nombre de sauvegarde
  - ID de la première sauvegarde
  - ID de la dernière sauvegarde
  - politique de rétention

Exemple de sortie de la commande :

```
$ barman status pgsrv
Server pgsrv:
  Description: PostgreSQL Instance pgsrv
  Active: True
  Disabled: False
  PostgreSQL version: 12.1
  Cluster state: in production
  pgespresso extension: Not available
  Current data size: 24.4 MiB
  PostgreSQL Data directory: /var/lib/pgsql/12/data
  Current WAL segment: 000000010000000000000004
  PostgreSQL 'archive_command' setting: barman-wal-archive localhost pgsrv %p
```

```
Last archived WAL: 000000010000000000000003, at Wed Dec 11 11:44:12 2019
Failures of WAL archiver: 52 (000000010000000000000001 at Wed Dec 11 11:44:04 2019)
Server WAL archiving rate: 1.41/hour
Passive node: False
Retention policies: not enforced
No. of available backups: 0
First available backup: None
Last available backup: None
Minimum redundancy requirements: satisfied (0/0)
```

3.4.24 Barman - Diagnostiquer

- La commande `diagnose` renvoie
  - les informations renvoyées par la commande `status`
  - des informations supplémentaires (sur le système par exemple)
  - au format `json`
- Exemple

```
$ sudo -u barman barman diagnose
```

La commande `diagnose` retourne les informations importantes concernant toutes les instances à sauvegarder, en donnant par exemple les versions de chacun des composants utilisés.

Elle reprend également les informations retournées par la commande `status`, le tout au format JSON.

3.4.25 Barman - Nouvelle sauvegarde

- Pour déclencher une nouvelle sauvegarde

```
$ sudo -u barman barman backup {<instance> | all} [--wait]
```

- Le détail de sauvegarde effectuée est affiché en sortie

La commande `backup` lance immédiatement une nouvelle sauvegarde, pour une seule instance si un identifiant est passé en argument, ou pour toutes les instances configurées si le mot-clé `all` est

utilisé.

L'option `--wait` permet d'attendre que les WAL soient archivés avant de rendre la main.

Exemple de sortie de la commande :

```
$ barman backup pgsrv
Starting backup using rsync-exclusive method for server pgsrv in
                                /var/lib/barman/pgsrv/base/20191211T121244
Backup start at LSN: 0/5000028 (000000010000000000000005, 00000028)
This is the first backup for server pgsrv
WAL segments preceding the current backup have been found:
  000000010000000000000001 from server pgsrv has been removed
  000000010000000000000002 from server pgsrv has been removed
  000000010000000000000003 from server pgsrv has been removed
Starting backup copy via rsync/SSH for 20191211T121244
Copy done (time: 1 second)
This is the first backup for server pgsrv
Asking PostgreSQL server to finalize the backup.
Backup size: 24.3 MiB. Actual size on disk: 24.3 MiB (-0.00% deduplication ratio).
Backup end at LSN: 0/5000138 (000000010000000000000005, 00000138)
Backup completed (start time: 2019-12-11 12:12:44.788598, elapsed time: 5 seconds)
Processing xlog segments from file archival for pgsrv
  000000010000000000000004
  000000010000000000000005
  000000010000000000000005.00000028.backup
```

3.4.26 Barman - Lister les sauvegardes

- Pour lister les sauvegardes existantes

```
$ sudo -u barman barman list-backup {<instance> | all}
```

- Affiche notamment la taille de la sauvegarde et des WAL associés

Liste les sauvegardes du catalogue, soit par instance, soit toutes si le mot-clé `all` est passé en argument.

Exemple de sortie de la commande :

```
$ barman list-backup pgsrv
pgsrv 20191211T121244 - Wed Dec 11 12:12:47 2019 - Size: 40.3 MiB -
                        WAL Size: 0 B
```

3.4.27 Barman - Détail d'une sauvegarde

- `show-backup` affiche le détail d'une sauvegarde (taille…)

```
$ sudo -u barman barman show-backup <instance> <ID-sauvegarde>
```

- `list-files` affiche le détail des fichiers d'une sauvegarde

```
$ sudo -u barman barman list-files <instance> <ID-sauvegarde>
```

La commande `show-backup` affiche toutes les informations relatives à une sauvegarde en particulier, comme l'espace disque occupé, le nombre de journaux de transactions associés, etc.

La commande `list-files` permet quant à elle d'afficher la liste complète des fichiers contenus dans la sauvegarde.

Exemple de sortie de la commande `show-backup` :

```
$ barman show-backup pgsrv 20191211T121244
Backup 20191211T121244:
  Server Name            : pgsrv
  System Id              : 6769104211696624889
  Status                 : DONE
  PostgreSQL Version     : 120001
  PGDATA directory       : /var/lib/pgsql/12/data

  Base backup information:
    Disk usage           : 24.3 MiB (40.3 MiB with WALs)
    Incremental size     : 24.3 MiB (-0.00%)
    Timeline             : 1
    Begin WAL            : 000000010000000000000005
    End WAL              : 000000010000000000000005
    WAL number           : 1
    Begin time           : 2019-12-11 12:12:44.526305+01:00
    End time             : 2019-12-11 12:12:47.794687+01:00
    Copy time            : 1 second + 1 second startup
    Estimated throughput : 14.3 MiB/s
    Begin Offset         : 40
    End Offset           : 312
    Begin LSN           : 0/5000028
    End LSN             : 0/5000138

  WAL information:
    No of files          : 0
    Disk usage           : 0 B
    Last available       : 000000010000000000000005

  Catalog information:
```

```
Retention Policy       : not enforced
Previous Backup        : - (this is the oldest base backup)
Next Backup            : - (this is the latest base backup)
```

3.4.28 Barman - Suppression d'une sauvegarde

- Pour supprimer manuellement une sauvegarde

```
$ sudo -u barman barman delete <instance> <ID-sauvegarde>
```

- Renvoie une erreur si la redondance minimale ne le permet pas

La suppression d'une sauvegarde nécessite de spécifier l'instance ciblée et l'identifiant de la sauvegarde à supprimer.

Cet identifiant peut être trouvé en utilisant la commande Barman `list-backup`.

Si le nombre de sauvegardes (après suppression) ne devait pas respecter le seuil défini par la directive `minimum_redundancy`, la suppression ne sera alors pas possible.

3.4.29 Barman - Conserver une sauvegarde

- Pour conserver une sauvegarde

```
$ sudo -u barman barman keep <instance> <ID-sauvegarde>
```

- Pour relâcher une sauvegarde

```
$ sudo -u barman barman keep --release <instance> <ID-sauvegarde>
```

Il est possible de marquer une sauvegarde pour qu'elle soit conservée par barman quelle que soit la rétention configurée avec la commande `barman keep <instance> <ID-sauvegarde>`.

La sauvegarde peut être relâchée en ajoutant le paramètre `--release`.

3.4.30 Barman - Tâches de maintenance

- La commande Barman `cron` déclenche la maintenance
  - récupération des WAL archivés
  - compression
  - politique de rétention
  - démarrage de `pg_receivewal`
- Exemple

```
$ sudo -u barman barman cron
```

- À planifier ! (vérifier `/etc/cron.d/barman`)

La commande `cron` permet d'exécuter les tâches de maintenance qui doivent être exécutées périodiquement, telles que l'archivage des journaux de transactions (déplacement du dossier `incoming_wals/` vers `wals/`), ou la compression.

L'application de la politique de rétention est également faite dans ce cadre.

Le démarrage de la commande `pg_recievewal` est aussi gérée par ce biais.

L'exécution de cette commande doit donc être planifiée via votre ordonnanceur préféré (`cron` d'Unix par exemple), par exemple toutes les minutes.

Si vous avez installé Barman via les paquets (rpm ou debian), une tâche `cron` exécutée toutes les minutes a été créée automatiquement.

3.4.31 Barman - Restauration

- Copie/transfert de la sauvegarde
- Copie/transfert des journaux de transactions
- Génère le paramétrage pour la restauration
- Copie/transfert des fichiers de configuration

Le processus de restauration géré par Barman reste classique, mais nécessite tout de même quelques points d'attention.

En particulier, les fichiers de configuration sauvegardés sont restaurés dans le dossier `$PGDATA`, ce n'est potentiellement pas le bon emplacement selon le type d'installation / configuration de l'instance. Dans une installation basée sur les paquets Debian/Ubuntu par exemple, les fichiers de configuration se trouvent dans `/etc/postgresql/<version>/<instance>` et non dans le répertoire PGDATA. Il convient donc de penser à les supprimer du `PGDATA` s'ils n'ont rien à y faire avant de démarrer l'instance.

De même, la directive de configuration `archive_command` est passée à `false` par Barman. Une fois l'instance démarrée et fonctionnelle, il convient de modifier la valeur de ce paramètre pour réactiver l'archivage des journaux de transactions.

3.4.32 Barman - Options de restauration

- Locale ou à distance
- Cibles : timeline, date, ID de transaction ou point de restauration
- Déplacement des tablespaces

Au niveau de la restauration, Barman offre la possibilité de restaurer soit en local (sur le serveur où se trouvent les sauvegardes), soit à distance.

Le cas le plus commun est une restauration à distance, car les sauvegardes sont généralement centralisées sur le serveur de sauvegarde d'où Barman est exécuté.

Pour la restauration à distance, Barman s'appuie sur la couche SSH pour le transfert des données.

Barman supporte différents types de cibles dans le temps pour la restauration :

- **timeline** : via l'option `--target-tli`, lorsqu'une divergence de timeline a eu lieu, il est possible de restaurer et rejouer toutes les transactions d'une timeline particulière ;
- **date** : via l'option `--target-time` au format **YYYY-MM-DD HH:MM:SS.mmm**, spécifie une date limite précise dans le temps au delà de laquelle la procédure de restauration arrête de rejouer les transactions ;
- **identifiant de transaction** : via l'option `--target-xid`, restauration jusqu'à une transaction précise ;
- **point de restauration** : via l'option `--target-name`, restauration jusqu'à un point de restauration créé préalablement sur l'instance via l'appel à la fonction `pg_create_restore_point(nom)`.

Barman permet également de relocaliser un tablespace lors de la restauration.

Ceci est utile lorsque l'on souhaite restaurer une sauvegarde sur un serveur différent, ne disposant pas des même points de montage des volumes que l'instance originelle.

3.4.33 Barman - Exemple de restauration à distance

- Exemple d'une restauration
  - déclenchée depuis le serveur Barman
  - avec un point dans le temps spécifié

```
$ sudo -u barman barman recover                  \
    --remote-ssh-command "ssh postgres@pgsrv"    \
    --target-time "2019-12-11 14:00:00"          \
    pgsrv 20191211T121244 /var/lib/pgsql/12/data/
```

Dans cet exemple, nous souhaitons effectuer une restauration à distance via l'option `--remote-ssh-command`, prenant en argument `"ssh postgres@pgsrv"` correspondant à la commande SSH pour se connecter au serveur à restaurer.

L'option `--target-time` définit ici le point de restauration dans le temps comme étant la date « 2019-12-11 14:00:00 ».

Les trois derniers arguments sont :

- l'identifiant de l'instance dans le fichier de configuration de Barman : `pgsrv` ;
- l'identifiant de la sauvegarde cible : `20191211T121244` ;
- et enfin le dossier PGDATA de l'instance à restaurer.

3.5 PITRERY - PRÉSENTATION GÉNÉRALE

- R&D Dalibo
- Langage : **bash**
- OS : **Unix/Linux**
- Versions compatibles : **8.2 à 14** (pas 15+)
- Développement arrêté, ne plus utiliser

pitrery est un outil de gestion de sauvegarde physique et restauration PITR, écrit en bash, issu du labo R&D de Dalibo.

Il est compatible avec tous les environnements Unix/Linux disposant de l'interpréteur de shell `bash`, et supporte toutes les versions de PostgreSQL depuis la 8.2 jusqu'à la version 14, qui est la dernière version supportée.

Après 10 ans de développement actif, le projet Pitrery est désormais placé en maintenance LTS (*Long Term Support*) jusqu'en novembre 2026. Plus aucune nouvelle fonctionnalité n'y sera ajoutée, les mises à jour concerneront les correctifs de sécurité uniquement. Il est désormais conseillé de lui préférer pgBackRest. Il n'est plus compatible avec PostgreSQL 15 et supérieur.

Site Web de pitrery[14].

[14] https://dalibo.github.io/pitrery/

3.6 AUTRES OUTILS DE L'ÉCOSYSTÈME

- De nombreux autres outils existent
  - …ou ont existé
- WAL-E, OmniPITR, pg_rman, walmgr…
- WAL-G

Du fait du dynamisme du projet, l'écosystème des outils autour de PostgreSQL est très changeant.

À côté des outils évoqués ci-dessus, que nous recommandons, on trouve de nombreux projets autour du thème de la gestion des sauvegardes.

Certains de ces projets répondent à des problématiques spécifiques, d'autres sont assez anciens et plus guère maintenus (comme WAL-E[15]), rendus inutiles par l'évolution de PostgreSQL ces dernières années (comme walmgr, de la suite Skytools[16], ou OmniPITR[17]) ou simplement peu actifs et peu rencontrés en production (par exemple pg_rman[18], développé par NTT).

Le plus intéressant et actif est sans doute WAL-G.

3.6.1 WAL-G - présentation

- Successeur de WAL-E, par Citus Data & Yandex
- Orientation cloud
- Aussi pour MySQL et SQL Server

WAL-G[19] est une réécriture d'un ancien outil assez populaire, WAL-E, par Citus[20] et Yandex, et actif.

De par sa conception, il est optimisé pour l'archivage des journaux de transactions vers des stockages *cloud* (Amazon S3, Google, Yandex), la compression multi-processeurs par différents algorithmes[21] et l'optimisation du temps de restauration. Il supporte aussi MySQL et SQL Server (et d'autres dans le futur).

[15] https://github.com/wal-e/wal-e
[16] https://wiki.postgresql.org/wiki/SkyTools
[17] https://github.com/omniti-labs/omnipitr
[18] https://github.com/ossc-db/pg_rman
[19] https://github.com/wal-g/wal-g
[20] https://www.citusdata.com/blog/2017/08/18/introducing-wal-g-faster-restores-for-postgres/
[21] https://wal-g.readthedocs.io/

3.7 CONCLUSION

- Des outils pour vous aider !
- Pratiquer, pratiquer et pratiquer
- Superviser les sauvegardes !

Nous venons de vous présenter des outils qui vont vous permettre de vous simplifier la tâche dans la mise en place d'une solution de sauvegarde fiable et robuste de vos instance PostgreSQL.

Cependant, leur maîtrise passera par de la pratique, et en particulier, la pratique de la restauration.

Le jour où la restauration d'une instance de production se présente, ce n'est généralement pas une situation confortable à cause du stress lié à une perte/corruption de données, interruption du service, etc. Autant maîtriser les outils qui vous permettront de sortir de ce mauvais pas.

N'oubliez pas également l'importance de la supervision des sauvegardes !

3.8 QUIZ

https://dali.bo/i4_quiz

4/ Solutions de réplication

Source de la photo : epSos.de[1] via Wikimedia[2], licence CC-BY-2.0.

[1] https://www.flickr.com/photos/36495803@N05/3574411866/
[2] https://commons.wikimedia.org/wiki/File:Green_Elephants_Garden_Sculptures.jpg

4.1 PRÉAMBULE

- Attention au vocabulaire !
- Identifier le besoin
- Keep It Simple…

La réplication est le processus de partage d'informations permettant de garantir la sécurité et la disponibilité des données entre plusieurs serveurs et plusieurs applications. Chaque SGBD dispose de différentes solutions pour cela et introduit sa propre terminologie. Les expressions telles que « cluster », « actif/passif » ou « primaire/secondaire » peuvent avoir un sens différent selon le SGBD choisi. Dès lors, il devient difficile de comparer et de savoir ce que désignent réellement ces termes. C'est pourquoi nous débuterons ce module par un rappel théorique et conceptuel. Nous nous attacherons ensuite à citer les outils de réplication, internes et externes.

4.1.1 Au menu

- Rappels théoriques
- Réplication interne
  - réplication physique
  - réplication logique
- Quelques logiciels externes de réplication
- Alternatives

Dans cette présentation, nous reviendrons rapidement sur la classification des solutions de réplication, qui sont souvent utilisés dans un but de haute disponibilité, mais pas uniquement.

PostgreSQL dispose d'une réplication physique basée sur le rejeu des journaux de transactions par un serveur dit « en *standby* ». Nous présenterons ainsi les techniques dites de *Warm Standby* et de *Hot Standby*.

Depuis la version 10 existe aussi une réplication logique, basée sur le transfert du résultat des ordres.

Nous détaillerons ensuite les projets de réplication autour de PostgreSQL les plus en vue actuellement.

4.1.2 Objectifs

- Identifier les différences entre les solutions de réplication proposées
- Choisir le système le mieux adapté à votre besoin

La communauté PostgreSQL propose plusieurs réponses aux problématiques de réplication. Le but de cette présentation est de vous apporter les connaissances nécessaires pour comparer chaque solution et comprendre les différences fondamentales qui les séparent.

À l'issue de cette présentation, vous serez capable de choisir le système de réplication qui correspond le mieux à vos besoins et aux contraintes de votre environnement de production.

4.2 RAPPELS THÉORIQUES

- Termes
- Réplication
  - synchrone / asynchrone
  - symétrique / asymétrique
  - diffusion des modifications

Le domaine de la haute disponibilité est couvert par un bon nombre de termes qu'il est préférable de définir avant de continuer.

4.2.1 Cluster, primaire, secondaire, *standby*…

- **Cluster** : ambiguïté !
  - groupe de bases de données = 1 instance (PostgreSQL)
  - groupe de serveurs (haute disponibilité et/ou réplication)
- Pour désigner les membres :
  - Primaire/*primary*
  - Secondaire/*standby*

Toute la documentation (anglophone) de PostgreSQL parle de *cluster* dans le contexte d'un serveur PostgreSQL seul. Dans ce contexte, le *cluster* est un groupe de bases de données, groupe étant la traduction directe de *cluster*. En français, on évitera tout ambiguïté en parlant d'« instance ».

Dans le domaine de la haute disponibilité et de la réplication, un *cluster* désigne un groupe de serveurs. Par exemple, un groupe d'un serveur primaire et de ses deux serveurs secondaires compose un cluster de réplication.

Le serveur, généralement unique, ouvert en écriture est désigné comme « **primaire** » (*primary*), parfois « principal ». Les serveurs connectés dessus sont « **secondaires** » (*secondary*) ou « *standby* » (prêts à prendre le relai). Les termes « maître/esclave » sont à éviter mais encore très courants. On trouvera dans le présent cours aussi bien « primaire/secondaire » que « principal/*standby* ».

4.2.2 Réplication asynchrone asymétrique

- **Asymétrique**
  - écritures sur un serveur primaire unique
  - lectures sur le primaire et/ou les secondaires
- **Asynchrone**
  - les écritures sur les serveurs secondaires sont différées
  - perte de données possible en cas de crash du primaire
- Quelques exemples
  - *streaming replication*, Slony, Bucardo

Dans la réplication asymétrique, seul le serveur primaire accepte des écritures, et les serveurs secondaires ne sont accessibles qu'en lecture.

Dans la réplication asynchrone, les écritures sont faites sur le primaire et le client reçoit une validation de l'écriture avant même qu'elles ne soient poussées vers le secondaire. La mise à jour des tables répliquées **est différée** (asynchrone). Le délai de réplication dépend de la technique et de la charge.

L'inconvénient de ce délai est que certaines données validées sur le primaire pourraient ne pas être disponibles sur les secondaires si le primaire est détruit avant que toutes les données, déjà validées auprès des clients, ne soient poussées sur les secondaires.

Autrement dit, il existe une fenêtre de temps, plus ou moins longue, de perte de données (qui entre dans le calcul du RPO).

4.2.3 Réplication asynchrone symétrique

- **Symétrique**
  - « multi-maîtres »
  - écritures sur les différents primaires
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents primaires
- **Asynchrone**
  - la réplication des écritures est différées
  - perte de données possible en cas de crash du serveur primaire
  - **risque d'incohérences !**
- Exemples :
  - BDR (EDB) : réplication logique
  - Bucardo : réplication par triggers

Dans la réplication symétrique, tous les serveurs sont accessibles aussi bien en lecture qu'en écriture. On parle souvent de « multi-maîtres ».

La réplication asynchrone, comme indiqué précédemment, envoie des modifications vers les autres membres du cluster mais n'attend aucune validation de leur part. Il y a donc toujours un risque de perte de données déjà validées si le serveur tombe sans avoir eu le temps de les répliquer vers au moins un autre serveur du cluster.

Ce mode de réplication ne respecte généralement pas les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) car si une copie échoue sur l'autre primaire alors que la transaction a déjà été validée, on peut alors arriver dans une situation où les données sont incohérentes entre les serveurs. Généralement, ce type de système doit proposer un gestionnaire de conflits, de préférence personnalisable, pour gérer ces cas de figure.

PostgreSQL ne supporte pas la réplication symétrique nativement. Plusieurs projets ont tenté de remplir ce vide.

BDR[3], de 2nd Quadrant (à présent EDB), se base sur la réplication logique et une extension propre au-dessus de PostgreSQL. BDR assure une réplication symétrique de toutes les données concernées de plusieurs instances toutes liées aux autres, et s'adresse notamment au cas de bases dont on a besoin dans des lieux géographiquement très éloignés. La gestion des conflits est automatique, mais les verrous ne sont pas propagés pour des raisons de performance, ce qui peut donc poser des problèmes d'intégrité, selon les options choisies. Les données peuvent différer entre les nœuds. L'application

[3] https://www.enterprisedb.com/docs/bdr/latest/

doit tenir compte de tout cela. Par exemple, il vaut mieux privilégier les UUID pour éviter les conflits. Les premières versions de BDR étaient sous licence libre, mais ne sont plus supportées, et la version actuelle (BDR4) est propriétaire et payante.

Bucardo[4] se base, lui, sur de la réplication par triggers. Il sera évoqué plus loin.

Si l'application le permet, il est possible de se rabattre sur un modèle où chaque instance est le point d'entrée unique de certaines données (par exemple selon la géographie), et les autres n'en ont que des copies en lecture, obtenues d'une manière ou d'une autre, ou doivent accéder au serveur responsable en écriture. Il s'agit alors plus d'une forme de *sharding* que de véritable réplication symétrique.

4.2.4 Réplication synchrone asymétrique

- **Asymétrique**
  - écritures sur un serveur primaire unique
  - lectures sur le serveur primaire et/ou les secondaires
- **Synchrone**
  - les écritures sur les secondaires sont immédiates
  - le client sait si sa commande a réussi sur plusieurs serveurs

Dans la réplication asymétrique, seul le serveur primaire accepte des écritures, les secondaires ne sont accessibles qu'en lecture.

Dans la réplication synchrone, le client envoie sa requête en écriture sur le serveur primaire, le serveur primaire l'écrit sur son disque, il envoie les données au serveur secondaire attend que ce dernier l'écrive sur son disque. Si tout ce processus s'est bien passé, le client est averti que l'écriture a été réalisée avec succès. Concrètement, un ordre `COMMIT` rend la main une fois l'écriture validée sur plusieurs serveurs, généralement au moins deux (un primaire et un secondaire). On utilise généralement un mécanisme dit de *Two Phase Commit* ou « validation en deux phases », qui assure qu'une transaction est validée sur tous les nœuds dans la même transaction. Les propriétés ACID sont dans ce cas respectées.

Le gros avantage de ce système est qu'il n'y a pas de risque de perte de données quand le serveur primaire s'arrête brutalement et qu'il est nécessaire de basculer sur un serveur secondaire. L'inconvénient majeur est que cela ralentit fortement les écritures.

PostgreSQL permet d'ajuster ce ralentissement à la criticité. Par défaut (paramètre `synchronous_commit` à `on`), la réplication synchrone garantit que le serveur secondaire a bien écrit la transaction dans ses journaux et qu'elle a été synchronisée sur son disque (`fsync`), en plus de celui du primaire

[4] https://www.bucardo.org/Bucardo/

bien sûr. En revanche, elle ne garantit pas forcément que le secondaire a bien rejoué la transaction : il peut se passer un laps de temps où une lecture sur le secondaire serait différente du serveur primaire (le temps que le secondaire rejoue la transaction). PostgreSQL dispose d'un mode (`synchronous_commit` à `remote_apply`) qui permet d'avoir la garantie que les modifications sont rejouées sur le secondaire, au prix d'une latence supplémentaire. À l'inverse, on peut estimer qu'il est suffisant que les données soient juste écrites dans la mémoire cache du serveur secondaire, et pas forcément sur disque, pour être considérées comme répliquées (`synchronous_commit` à `remote_write`). La synchronicité peut être désactivée pour les requêtes peu critiques (`synchronous_commit` à `local`, voire `off`). Ce paramétrage peut être ajusté selon les besoins, requête par requête.

PostgreSQL permet aussi de disposer de plusieurs secondaires synchrones.

4.2.5 Réplication synchrone symétrique

- **Symétrique**
  - écritures sur les différents serveurs primaires
  - besoin d'un gestionnaire de conflits
  - lectures sur les différents serveurs
- **Synchrone**
  - les écritures sur les autres serveurs sont immédiates
  - le client sait si sa commande est validée sur plusieurs serveurs
  - risque important de lenteur !

Ce système est le plus intéressant… en théorie. L'utilisateur peut se connecter à n'importe quel serveur pour des lectures et des écritures. Il n'y a pas de risque de perte de données, vu que la commande ne réussit que si les données sont bien enregistrées sur tous les serveurs. Autrement dit, c'est le meilleur système de réplication et de répartition de charge.

Dans les inconvénients, il faut gérer les éventuels conflits qui peuvent survenir quand deux transactions concurrentes opèrent sur le même ensemble de lignes. On résout ces cas particuliers avec des algorithmes plus ou moins complexes. Il faut aussi accepter la perte de performance en écriture induite par le côté synchrone du système.

PostgreSQL ne supporte pas la réplication symétrique, donc encore moins la symétrique synchrone. Certains projets évoqués essaient ou ont essayé d'apporter cette fonctionnalité.

Le besoin d'une architecture « multi-maîtres » revient régulièrement, mais il faut s'assurer qu'il est réel. Avant d'envisager une architecture complexe, et donc source d'erreurs, optimisez une installation asy-

métrique simple et classique, quitte à multiplier les serveurs secondaires, et testez ses performances : PostgreSQL pourrait bien vous surprendre !

Selon les cas d'utilisation, la réplication logique peut aussi être utile.

4.2.6 Diffusion des modifications

- Par requêtes
  - diffusion de la requête
- Par triggers
  - diffusion des données résultant de l'opération
- Par journaux, physique
  - diffusion des blocs disques modifiés
- Par journaux, logique
  - extraction et diffusion des données résultant de l'opération depuis les journaux

La récupération des données de réplication se fait de différentes façons suivant l'outil utilisé.

La diffusion de l'opération de mise à jour (donc **le SQL lui-même**) est très flexible et compatible avec toutes les versions. Cependant, cela pose la problématique des opérations dites non déterministes. L'insertion de la valeur `now()` exécutée sur différents serveurs fera que les données seront différentes, généralement très légèrement différentes, mais différentes malgré tout. L'outil Pgpool, qui implémente cette méthode de réplication, est capable de récupérer l'appel à la fonction `now()` pour la remplacer par la date et l'heure. En effet, il connaît les différentes fonctions de date et heure proposées en standard par PostgreSQL. Cependant, il ne connaît pas les fonctions utilisateurs qui pourraient faire de même. Il est donc préférable de renvoyer les données, plutôt que les requêtes.

Le renvoi du résultat peut se faire de deux façons : soit en récupérant les nouvelles données avec un trigger, soit en récupérant les nouvelles données dans les journaux de transactions.

Cette première solution est utilisée par un certain nombre d'outils externes de réplication, comme Slony ou Bucardo. Les fonctions triggers étant écrites en C, cela assure de bonnes performances. Cependant, seules les modifications des données sont attrapables avec des triggers, les modifications de la structure ne sont généralement pas gérées. Autrement dit, l'ajout d'une table, l'ajout d'une colonne demande une administration plus poussée, non automatisable.

La deuxième solution (par journaux de transactions) est bien plus intéressante car les journaux contiennent toutes les modifications, données comme structures. Il suffit au secondaire de réappli-

quer tous les journaux provenant du primaire pour être à l'image exacte de celui-ci. De ce fait, une fois mise en place, cette méthode requiert peu de maintenance. PostgreSQL offre nativement depuis longtemps deux variantes de cette solution : par journaux entiers (*log shipping*) ou par flux (*streaming replication*).

PostgreSQL permet, depuis la version 10, le décodage logique des modifications de données correspondant aux blocs modifiés dans les journaux de transactions. L'objectif est de permettre l'extraction logique des données écrites permettant la mise en place d'une réplication logique des résultats entièrement intégrée, donc sans triggers. Les modifications de structures doivent être gérées à la main.

4.3 RÉPLICATION INTERNE PHYSIQUE

- Réplication
  - asymétrique
  - asynchrone (défaut) ou synchrone (et selon les transactions)
- Secondaires
  - non disponibles (*Warm Standby*)
  - disponibles en lecture seule (*Hot Standby*)
  - cascade
  - retard programmé

La réplication physique de PostgreSQL est par défaut **asynchrone** et **asymétrique**. Il est possible de sélectionner le mode synchrone/asynchrone pour chaque serveur secondaire individuellement, et séparément pour chaque transaction, en modifiant le paramètre `synchronous_commit`.

La réplication physique fonctionne par l'envoi des enregistrements des journaux de transactions, soit par envoi de fichiers complets (on parle de *log shipping*), soit par envoi de groupes d'enregistrements en flux (on parle là de *streaming replication*), puisqu'il s'agit d'une réplication par diffusion de journaux.

La différence entre *Warm Standby* et *Hot Standby* est très simple :

- un serveur secondaire en *Warm Standby* est un serveur de secours sur lequel il n'est pas possible de se connecter ;
- un serveur secondaire en *Hot Standby* accepte les connexions et permet l'exécution de requêtes en lecture seule.

Un secondaire peut récupérer les informations de réplication depuis un autre serveur secondaire (fonctionnement en cascade).

Le serveur secondaire peut aussi n'appliquer les informations de réplication qu'après un délai configurable.

4.3.1 Log Shipping

- But :
  - envoyer les journaux de transactions à un secondaire
- Première solution disponible
- Gros inconvénients :
  - perte possible de plusieurs journaux
  - latence à la réplication
  - penser à `archive_timeout` ou `pg_receivewal`

Le *log shipping* permet d'envoyer les journaux de transactions terminés sur un autre serveur. Ce dernier peut être un serveur secondaire, en *Warm Standby* ou en *Hot Standby*, prêt à les rejouer.

Cependant, son gros inconvénient vient du fait qu'il faut attendre qu'un journal soit complètement écrit pour qu'il soit propagé vers le secondaire. Or, un journal de 16 Mo peut contenir plusieurs centaines de transactions ! Si l'archivage a du retard (grosse charge, réseau saturé…), plusieurs journaux peuvent même être en attente. Le retard du secondaire par rapport au primaire peut donc devenir important, ce qui est gênant dans le cas d'un *standby* utilisé en lecture seule, par exemple dans le cadre d'une répartition de la charge de lecture.

En conséquence il est possible de perdre toutes les transactions contenues dans le journal de transactions en cours, voire tous ceux en retard, en cas de *failover* et de destruction physique des journaux sur le primaire.

On peut cependant moduler le risque de trois façons:

- sauf avarie très grave sur le serveur primaire, les journaux de transactions en attente peuvent être récupérés et appliqués sur le serveur secondaire ;
- on peut réduire la fenêtre temporelle de la réplication en modifiant la valeur de la clé de configuration `archive_timeout`. Au delà du délai exprimé avec cette variable de configuration, le serveur change de journal de transactions, provoquant l'archivage du précédent. On peut par exemple envisager un `archive_timeout` à 30 secondes, et ainsi obtenir une réplication à 30 secondes près. Attention toutefois, les journaux archivés font toujours 16 Mo, qu'ils soient archivés remplis ou non ;
- on peut utiliser l'outil `pg_receivewal` (nommé `pg_receivexlog` jusqu'en 9.6) qui crée à distance les journaux de transactions à partir d'un flux de réplication.

4.3.2 Streaming replication

- But
  - avoir un retard moins important sur le serveur secondaire
- Rejouer **les enregistrements de transactions** du serveur primaire par **paquets**
  - paquets plus petits qu'un journal de transactions

L'objectif du mécanisme de la *streaming replication* est d'avoir un secondaire qui accuse moins de retard. En effet, comme on vient de le voir, le *log shipping* exige d'attendre qu'un journal soit complètement rempli avant qu'il ne soit envoyé au serveur secondaire.

La réplication par *streaming* diminue ce retard en envoyant les enregistrements des journaux de transactions par groupe bien inférieur à un journal complet. Il introduit aussi deux processus gérant le transfert du contenu des WAL entre le serveur primaire et le serveur secondaire. Ce flux est totalement indépendant de l'archivage du WAL. Ainsi, en cas de perte du serveur primaire, sauf retard à cause d'une saturation quelconque, la perte de données est très faible.

Les délais de réplication entre le serveur primaire et le serveur secondaire sont très courts : une modification sur le serveur primaire sera en effet très rapidement répliqué sur un secondaire.

C'est une solution éprouvée et au point depuis des années. Néanmoins, elle a ses propres inconvénients : réplication de l'instance complète, architecture matérielle et version majeure de PostgreSQL forcément identiques entre les serveurs du cluster, etc.

4.3.3 *Warm Standby*

- Serveur de secours
  - prêt à prendre le relai du primaire
  - (presque) identique au primaire
- Différentes configurations selon les versions
  - asynchrone ou synchrone
  - application immédiate ou retardée
- En pratique, préférer le *Hot Standby*

Le *Warm Standby* existe depuis la version 8.2. La robustesse de ce mécanisme simple est prouvée depuis longtemps.

Les journaux de transactions sont répliqués en *log shipping* ou *streaming replication* selon la version, le besoin et les contraintes d'architecture. Le serveur secondaire est en mode *recovery* perpétuel et applique automatiquement les journaux de transaction reçus.

Un serveur en *Warm Standby* n'accepte aucune connexion entrante. Il n'est utile que comme réplicat prêt à être promu en production à la place de l'actuel primaire en cas d'incident. Les serveurs secondaires sont donc généralement paramétrés directement en *Hot Standby*.

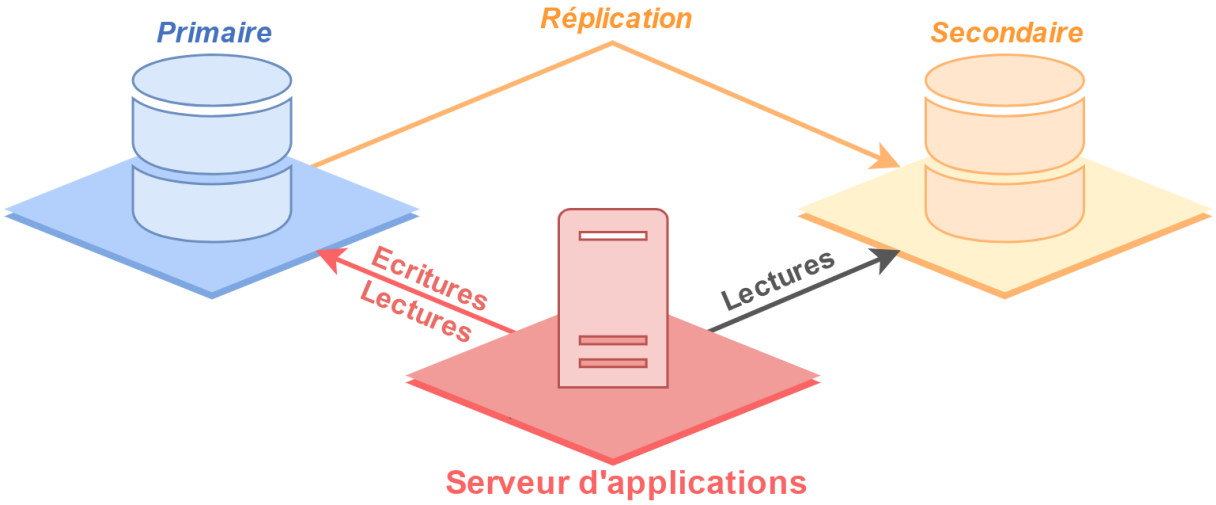
4.3.4 *Hot Standby*

- Serveur secondaire
  - accepte les connexions entrantes
  - requêtes en lecture seule et sauvegardes
  - prêt à prendre le relai du primaire
- Différentes configurations selon les versions
  - asynchrone ou synchrone
  - application immédiate ou retardée

Le *Hot Standby* est une évolution du *Warm Standby* : il accepte les connexions des utilisateurs et permet d'exécuter des requêtes en lecture seule.

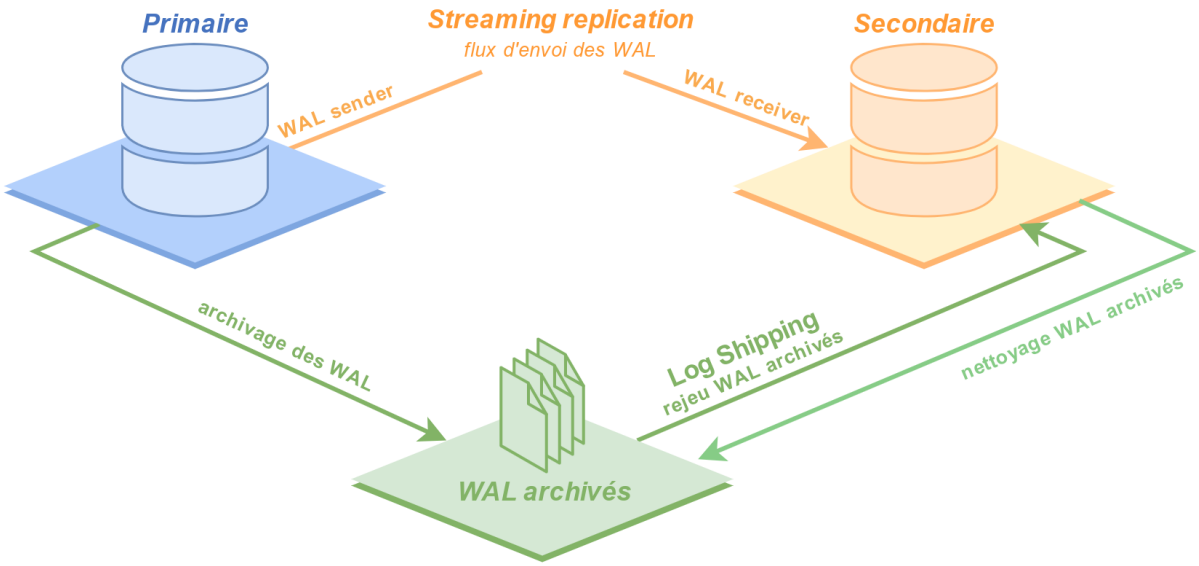
Ce serveur peut toujours remplir le rôle de serveur de secours, tout en étant utilisable pour soulager le primaire : sauvegarde, requêtage en lecture…

4.3.5 Exemple

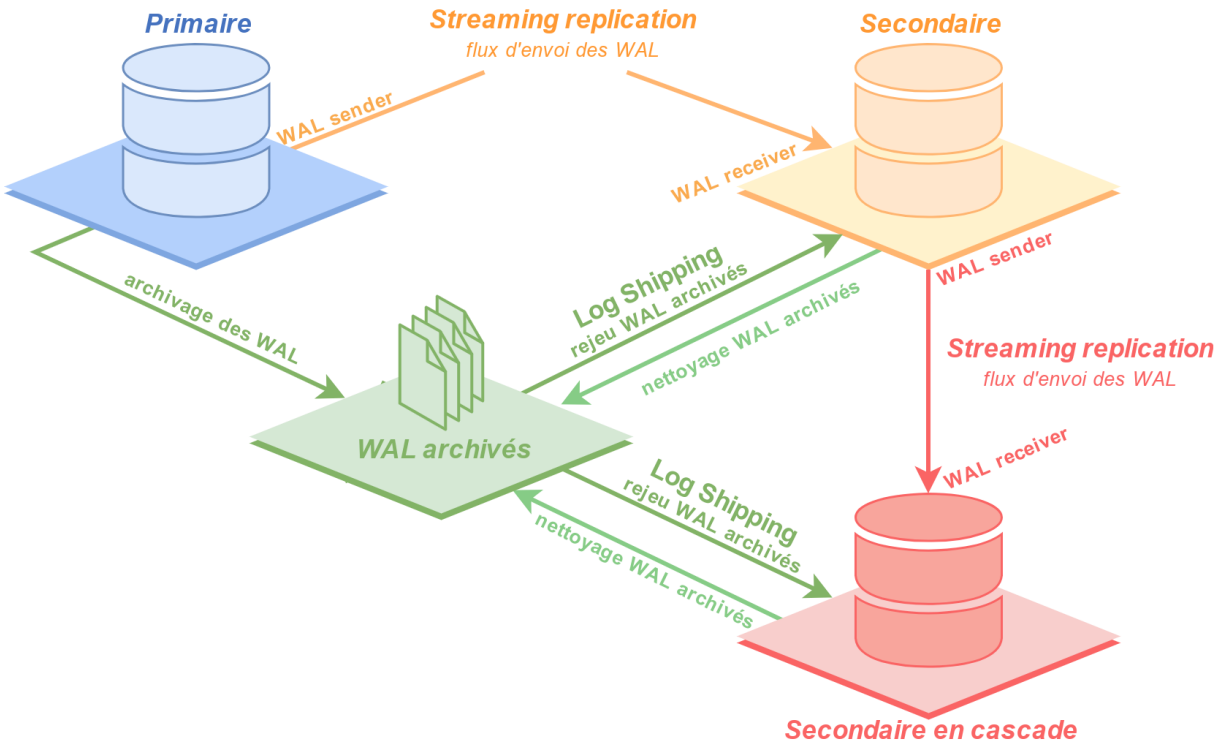


Cet exemple montre un serveur primaire en *streaming replication* vers un serveur secondaire. Ce dernier est configuré en *Hot Standby*. Ainsi, les utilisateurs peuvent se connecter sur le serveur secondaire pour les requêtes en lecture et sur le primaire pour des lectures comme des écritures. Cela permet une forme de répartition de charge des lectures, la répartition étant gérée par le serveur d'applications ou par un outil spécialisé.

4.3.6 Réplication interne



4.3.7 Réplication en cascade



4.4 RÉPLICATION INTERNE LOGIQUE

- Réplique les changements
  - d'une seule base de données
  - d'un ensemble de tables défini
- Principe Éditeur/Abonnés

Contrairement à la réplication physique, la réplication logique ne réplique pas les blocs de données. Elle décode le **résultat** des requêtes qui sont transmis au secondaire. Celui-ci applique les modifications issues du flux de réplication logique.

La réplication logique utilise un système de publication/abonnement avec un ou plusieurs « abonnés » qui s'abonnent à une ou plusieurs « publications » d'un nœud particulier.

Une publication peut être définie sur n'importe quel serveur primaire. Le nœud sur laquelle la publication est définie est nommé « éditeur ». Le nœud où un abonnement a été défini est nommé « abonné ».

Une publication est un ensemble de modifications générées par une table ou un groupe de table. Chaque publication existe au sein d'une seule base de données.

Un abonnement définit la connexion à une autre base de données et un ensemble de publications (une ou plus) auxquelles l'abonné veut souscrire.

La réplication logique est disponible depuis la version 10 de PostgreSQL. Contrairement à la réplication physique, elle s'effectue entre instances primaires, toutes deux ouvertes en écriture avec leurs tables propres. Rien n'interdit à une instance abonnée pour certaines tables de proposer ses propres publications, même de tables concernées par un abonnement.

4.4.1 Réplication logique - Fonctionnement

- Création d'une publication sur un serveur
- Souscription d'un autre serveur à cette publication
- Limitations :
  - DDL, Large objects, séquences, tables étrangères et vues matérialisées non répliqués
  - peu adaptée pour un *failover*

Une « publication » est créée sur le serveur éditeur et ne concerne que certaines tables. L'abonné souscrit à cette publication, c'est un « souscripteur ». Un processus spécial est lancé : le *logical replication worker*. Il va se connecter à un slot de réplication sur le serveur éditeur. Ce dernier va procéder à un décodage logique de ses propres journaux de transaction pour extraire les résultats des ordres SQL (et non les ordres eux-mêmes). Le flux logique est transmis à l'abonné qui les applique sur les tables.

La réplication logique possède quelques limitations. La principale est que seules les données sont répliquées, c'est-à-dire le résultat des ordres `INSERT`, `DELETE`, `UPDATE`, `TRUNCATE` (sauf en v10 pour ce dernier). Les tables cible doivent être créés manuellement, et il faudra dès lors répliquer manuellement les changements de structure.

Il n'est pas obligatoire de conserver strictement la même structure des deux côtés. Mais, afin de conserver sa cohérence, la réplication s'arrêtera en cas de conflit. Des clés primaires sur toutes les tables concernées sont fortement conseillées. Les *large objects* ne sont pas répliqués. Les séquences non plus, y compris celles utilisées par les colonnes de type `serial`. Notez que pour éviter des effets de bord sur la cible, les triggers des tables abonnées ne seront pas déclenchés par les modifications reçues via la réplication.

En principe, il serait possible d'utiliser la réplication logique en vue d'un *failover* vers un serveur de secours en propageant manuellement les mises à jour de séquences et de schéma. La réplication physique est cependant plus appropriée et plus efficace pour cela.

La réplication logique vise d'autres objectifs, tels la génération de rapports sur une instance séparée ou la migration vers une version majeure de PostgreSQL avec une indisponibilité minimale.

4.5 RÉPLICATION EXTERNE

- Outils les plus connus :
  - Pgpool
  - Slony, Bucardo
  - pgLogical
- Niches

Jusqu'en 2010, PostgreSQL ne disposait pas d'un système de réplication évolué, et plus d'une quinzaine de projets ont tenté de combler ce vide. L'arrivée de la réplication logique en version 10 met à mal les derniers survivants. En pratique, ces outils ne comblent plus que des niches.

La liste exhaustive est trop longue pour que l'on puisse évoquer en détail chacune des solutions, surtout que la plupart sont obsolètes ou ne semblent plus maintenues. Voici les plus connues :

- Slony (réplication par trigger, éprouvé et fiable) ;
- Bucardo (réplication par trigger, toujours maintenu) ;
- Pgpool (réplication d'ordres SQL parmi d'autres fonctionnalités ; toujours actif) ;
- pgLogical (réplication logique, toujours actif) ;
- Londiste ;
- PGCluster ;
- Mammoth Replicator/Replicator ;
- Daffodil Replicator ;
- RubyRep ;
- pg_comparator ;
- Postgres-X2 (ex-Postgres-XC)…

Pour les détails sur ces outils et d'autres, voir le wiki[5] ou cet article : Haute disponibilité avec PostgreSQL[6], Guillaume Lelarge, 2009.

[5] https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling
[6] https://public.dalibo.com/exports/conferences/_archives/_2011/201102_haute_disponibilite_avec_postgresql/solutionsha.pdf

4.6 SHARDING

- Répartition des données sur plusieurs instances
- Évolution horizontale en ajoutant des serveurs
- Parallélisation
- Clé de répartition cruciale
- Administration complexifiée
- Sous PostgreSQL :
  - *Foreign Data Wrapper*
  - PL/Proxy
  - Citus (extension), et nombreux *forks*

Le *sharding* n'est pas de la réplication, ce serait même l'inverse. Le principe consiste à répartir les données sur plusieurs instances différentes, chacune étant responsable d'une partie des données, et ouverte en écriture.

La volumétrie peut augmenter, il suffit de rajouter des serveurs. Plusieurs serveurs peuvent travailler en parallèle sur la même requête. On contourne ainsi le principal goulet d'étranglement des performances : les entrées-sorties.

Le problème fondamental est la clé de répartitions des données sur les différents serveurs. Un cas simple est la répartition des données de nombreux clients dans plusieurs instances, chaque client n'étant présent que dans une seule instance. On peut aussi opérer une sorte de *hash* de la clé pour répartir équitablement les données sur les serveurs. Il faut aviser en fonction de la charge prévue, de la nécessité d'éviter les conflits lors des mises à jour, du besoin de croiser les données en masse, des purges à faire de temps à autre, et de la manière de répartir harmonieusement les écritures et lectures entre ces instances. C'est au client ou à une couche d'abstraction de savoir quel(s) serveur(s) interroger.

PostgreSQL n'implémente pas directement le *sharding*. Plusieurs techniques et outils permettent de le mettre en place.

- Les *Foreign Data Wrappers* (et l'extension `postgres_fdw` en particulier) permettent d'accéder à des données présentes sur d'autres serveurs. La capacité de `postgres_fdw` à « pousser » filtres et jointures vers les serveurs distants s'améliore de version en version. Des tables distantes peuvent être montées en tant que partitions d'une table mère. Les insertions dans une table partitionnée peuvent même être redirigées vers une partition distante de manière transparente. Vous trouverez un exemple dans cet article[7] ou à la fin du module de formation V1[8] Le

[7] https://pgdash.io/blog/postgres-11-sharding.html
[8] https://dali.bo/v1_html#tables-distantes-sharding

parcours simultané des partitions distantes est même possible à partir de PostgreSQL 14[9]. La réplication logique peut synchroniser des tables non distribuées sur les instances.

- PL/Proxy est une extension qui permet d'appeler plusieurs hôtes distants à la fois avec un seul appel de fonctions. Elle existe depuis des années. Son inconvénient majeur est la nécessité de réécrire tous les appels à distribuer par des fonctions, on ne peut pas se reposer sur le SQL de manière transparente.
- Citus[10] est une extension libre dont le but est de rendre le *sharding* transparent, permettant de garder la compatibilité avec le SQL habituel. Des nœuds sont déclarés auprès du serveur principal (où les clients se connectent), et quelques fonctions permettent de déclarer les tables comme distribuées selon une clé (et découpées entre les serveurs) ou tables de références (dupliquées partout pour faciliter les jointures). Les requêtes sont redirigées vers le bon serveur selon la clé, ou réellement parallélisées sur les différents serveurs concernés. Le projet est vivant, bien documenté, et a bonne réputation. Depuis son rachat par Microsoft en 2019, Citus assure ses revenus grâce à une offre disponible sur le cloud Azure. Ceci a permis en 2022 de libérer les fonctionnalités payantes et de publier l'intégralité projet en open-source.

Le *sharding* permet d'obtenir des performances impressionnantes, mais il a ses inconvénients :

- plus de serveurs implique plus de sources de problèmes, de supervision, de tâches d'administration (chaque serveur a potentiellement son secondaire, sa réplication…) ;
- le modèle de données doit être adapté au problème, limitant les interactions d'un nœud avec les autres ; le choix de la clé est crucial ;
- les propriétés ACID et la cohérence sont plus difficilement respectées dans ces environnements.

Historiquement, plusieurs *forks* de PostgreSQL ont été développés en partie pour faire du *sharding*, principalement en environnement OLAP/décisionnel, comme PostgreSQL-XC/XL[11], à présent disparu, ou Greenplum[12], qui existe toujours. Ces *forks* ont plus ou moins de difficultés à suivre la rapidité d'évolution de la version communautaire : les choisir implique de se passer de certaines nouveautés. D'où le choix de Citus de la forme d'une extension.

Ce domaine est l'un de ceux où PostgreSQL devrait beaucoup évoluer dans les années à venir, comme le décrivait Bruce Momjian en septembre 2018[13].

[9] https://dali.bo/workshop14_html#lecture-asynchrone-des-tables-distantes
[10] https://www.citusdata.com/product
[11] https://www.postgres-xl.org/
[12] https://greenplum.org/
[13] https://momjian.us/main/writings/pgsql/sharding.pdf

4.7 RÉPLICATION BAS NIVEAU

- RAID
- DRBD
- SAN Mirroring
- À prendre évidemment en compte…

Même si cette présentation est destinée à détailler les solutions logicielles de réplication pour PostgreSQL, on peut tout de même évoquer les solutions de réplication de « bas niveau », voire matérielles.

De nombreuses techniques matérielles viennent en complément essentiel des technologies de réplication utilisées dans la haute disponibilité. Leur utilisation est parfois incontournable, du RAID en passant par les SAN et autres techniques pour redonder l'alimentation, la mémoire, les processeurs, etc.

4.7.1 RAID

- Obligatoire
- Fiabilité d'un serveur
- RAID 1 ou RAID 10
- RAID 5 déconseillé (performances)
- Lectures plus rapides
    - dépend du nombre de disques impliqués

Un système RAID 1 ou RAID 10 permet d'écrire les mêmes données sur plusieurs disques en même temps. Si un disque meurt, il est possible d'utiliser l'autre disque pour continuer. C'est de la réplication bas niveau. Le disque défectueux peut être remplacé sans interruption de service. Ce n'est pas une réplication entre serveurs mais cela contribue à la haute-disponibilité du système.

Le RAID 5 offre les mêmes avantages en gaspillant moins d'espace qu'un RAID 1, mais il est déconseillé pour les bases de données (PostgreSQL comme ses concurrents) à cause des performances en écriture, au quotidien comme lors de la reconstruction d'une grappe après remplacement d'un disque.

Le système RAID 10 est plus intéressant pour les fichiers de données alors qu'un système RAID 1 est suffisant pour les journaux de transactions.

Le RAID 0 (répartition des écritures sur plusieurs disques sans redondance) est évidemment à proscrire.

4.7.2 DRBD

- Simple / synchrone / Bien documenté
- Lent / Secondaire inaccessible / Linux uniquement

DRBD est un outil capable de répliquer le contenu d'un périphérique blocs. En ce sens, ce n'est pas un outil spécialisé pour PostgreSQL contrairement aux autres outils vus dans ce module. Il peut très bien servir à répliquer des serveurs de fichiers ou de mails. Il réplique les données en temps réel et de façon transparente, pendant que les applications modifient leur fichiers sur un périphérique. Il peut fonctionner de façon synchrone ou asynchrone. Tout ça en fait donc un outil intéressant pour répliquer le répertoire des données de PostgreSQL.

Pour plus de détails, consulter cet article[14] de Guillaume Lelarge dans Linux Magazine Hors Série 45.

DRBD est un système simple à mettre en place. Son gros avantage est la possibilité d'avoir une réplication synchrone. Ses inconvénients sont sa lenteur, la non-disponibilité des secondaires et un volume de données plus important à répliquer (WAL + tables + index + vues matérialisées…).

4.7.3 SAN Mirroring

- Comparable à DRBD
- Solution intégrée
- Manque de transparence

La plupart des constructeurs de baie de stockage proposent des systèmes de réplication automatisés avec des mécanismes de *failover*/*failback* parfois sophistiqués. Ces solutions présentent peu ou prou les mêmes caractéristiques, avantages et inconvéniants que DRBD. Ces technologies ont en revanche le défaut d'être opaques et de nécessiter une main d'œuvre hautement qualifiée.

4.8 CONCLUSION

Quelle que soit la solution envisagée :

- Bien définir son besoin
- Identifier tous les *SPOF*
- Superviser son *cluster*
- Tester régulièrement les procédures de *failover* (Loi de Murphy…)

Bibliographie :

- « Haute disponibilité, répartition de charge et réplication[15] » (documentation officielle)

4.8.1 Questions

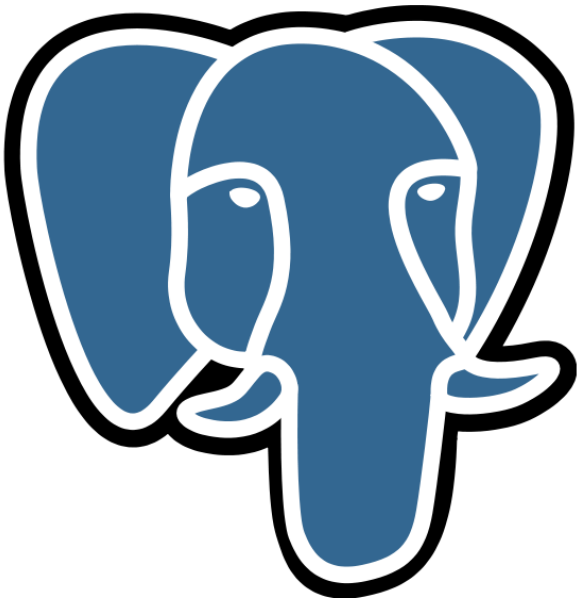
N'hésitez pas, c'est le moment !

[15] https://docs.postgresql.fr/current/high-availability.html

4.9 QUIZ

https://dali.bo/w1_quiz

5/ Réplication physique : fondamentaux



5.1 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.

5.1.1 Objectifs

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

5.2 CONCEPTS / PRINCIPES

- 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.

5.2.1 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`.

5.2.2 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.

5.2.3 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).

5.3 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.

5.3.1 Serveur primaire (1/2) - Configuration

Dans `postgresql.conf` :

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

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

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

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

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

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

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

5.3.2 Serveur primaire (2/2) - Authentification

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

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

- Recharger la configuration

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

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

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

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

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

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

et bien sûr un mot de passe complexe.

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

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

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

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

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

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

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

pg_basebackup :

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

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

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

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

Outils PITR :

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

rsync :

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

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

Rappelons les trois étapes essentielles :

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

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

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

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

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

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

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

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

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

5.3.5 Serveur secondaire (2/4) - Paramètres

- `primary_conninfo` (*streaming*) :

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3.6 Serveur secondaire (3/4) - Démarrage

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

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

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

5.3.7 Processus

Sur le primaire :

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

Sur le secondaire :

- `walreceiver   streaming 0/3BD48728`

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

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

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

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

5.4 PROMOTION

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

5.4.1 Attention au split-brain !

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

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

Quelques histoires « d'horreur » à ce sujet :

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

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

5.4.2 Vérification avant promotion

- Primaire :

```
# systemctl stop postgresql-14

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

- Secondaire :

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

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

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

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

5.4.3 Promotion du standby : méthode

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

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

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

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

5.4.4 Promotion du standby : déroulement

Une promotion déclenche :

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

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

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

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

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

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

```
recovery_target_timeline = latest
```

5.4.5 Opérations après promotion du standby

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

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

5.4.6 Retour à l'état stable

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

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

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

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

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

5.4.7 Retour à l'état stable, suite

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

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

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

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

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

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

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

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

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

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

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

5.5 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.

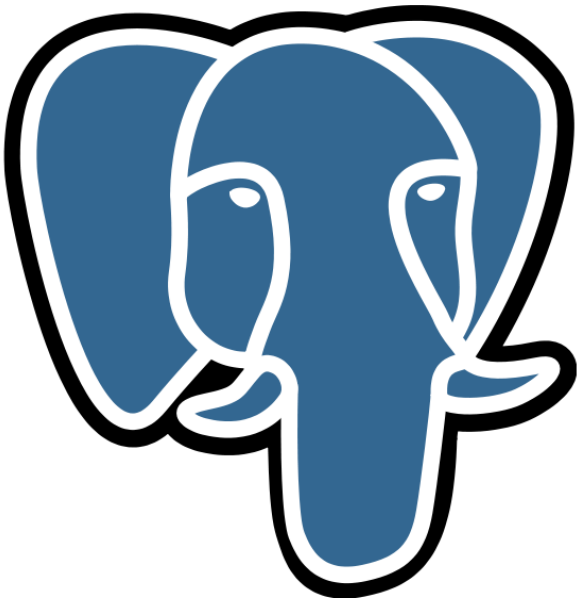
Cette réplication reste fidèle aux principes du moteur de PostgreSQL :

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

5.6 QUIZ

https://dali.bo/w2a_quiz

6/ Réplication physique avancée



6.1 INTRODUCTION

- Supervision
- Fonctionnalités avancées

En complément du module précédent, il est important de bien superviser un cluster en réplication, d'en connaître les limites mais aussi d'appréhender toutes les possibilités offertes par la réplication physique.

6.1.1 Au menu

- Supervision
- Gestion des conflits
- Asynchrone ou synchrone
- Réplication en cascade
- Slot de réplication
- Log shipping

6.2 SUPERVISION (STREAMING)

- Quelles vues et fonctions utilitaires ?
- Comment voir et calculer le retard des secondaires ?

6.2.1 Utilitaires pour le *streaming*

- `pg_is_in_recovery()` : instance en réplication ?
- Calcul du retard en octets :

```
-- primaire
SELECT pg_wal_lsn_diff ( pg_current_wal_lsn(), '0/73D3C1F0' );
```

- et en temps

```
-- secondaire
SELECT now() - pg_last_xact_replay_timestamp() ; -- si activité
```

Étant donné qu'il est possible de se connecter sur le serveur primaire comme sur le serveur secondaire, il est important de pouvoir savoir sur quel type de serveur un utilisateur est connecté. Pour cela, il existe une fonction appelée `pg_is_in_recovery()` : elle renvoie la valeur `true` si l'utilisateur se trouve sur un serveur en *hot standby* et `false` sinon.

Retard en octets :

Le calcul de la différence de volumétrie de données entre le primaire et ses secondaires (*lag*) peut-être effectué avec la fonction `pg_wal_lsn_diff()`. La fonction `pg_current_wal_lsn()` fournit la position dans le flux de données du primaire. En récupérant la position sur le secondaire (au choix, dernière réception avec `pg_last_wal_receive_lsn()` ou dernier rejeu avec `pg_last_wal_replay_lsn()`), le calcul du *lag* en octet devient :

```
-- sur le secondaire
SELECT  pg_last_wal_replay_lsn () ;

 pg_last_wal_replay_lsn
------------------------
 13/A7DD670

-- sur le primaire
SELECT pg_size_pretty (
```

```
  pg_wal_lsn_diff( pg_current_wal_lsn(), '13/A7DD670')
  ) ;

 pg_size_pretty
----------------
 1939 kB
```

Mais nous allons voir qu'il y a plus pratique.

Retard en durée :

Quand le retard d'un serveur secondaire sur son primaire est exprimé en octets, il n'est pas simple d'en appréhender l'amplitude. Le retard en durée est plus parlant. La fonction `pg_last_xact_replay_timestamp()` indique la date et l'heure de la dernière transaction rejouée. Soustraire la date et l'heure actuelle à cette fonction permet d'avoir une estimation sur le retard au rejeu d'un serveur secondaire sous la forme d'une durée.

```
SELECT now() - pg_last_xact_replay_timestamp() ; -- si activité
```

Attention, si le primaire ne reçoit que des transactions en lecture, le flux de journaux n'est pas forcément complètement vide, mais `pg_last_xact_replay_timestamp()` ne s'incrémente alors pas sur le secondaire ! `now() - pg_last_xact_replay_timestamp()` donnera alors une durée croissante dans le temps, même si le serveur secondaire n'a aucun retard.

6.2.2 pg_stat_replication

Type de réplication & *lag* des secondaires :

```
SELECT * FROM pg_stat_replication ;
-[ RECORD 1 ]----+------------------------------
pid              | 286511
usesysid         | 10
usename          | postgres
application_name | secondaire2
client_addr      | 192.168.0.55
client_hostname  |
client_port      | -1
backend_start    | 2023-12-19 10:41:47.431471+01
backend_xmin     |
state            | streaming
sent_lsn         | 14/C402A000
write_lsn        | 14/C402A000
flush_lsn        | 14/C402A000
replay_lsn       | 14/C311D460
write_lag        | 00:00:00.032183
flush_lag        | 00:00:00.032601
replay_lag       | 00:00:02.984354
sync_priority    | 1
sync_state       | sync
reply_time       | 2023-12-19 11:05:37.903584+01
```

Pour connaître l'état des différents serveurs secondaires connectés au serveur primaire, le plus simple est de récupérer les informations provenant de la vue `pg_stat_replication` du primaire. Elle permet de connaître l'état de tous les serveurs secondaires connectés en *streaming* (mais pas ceux déconnectés !). Il y a une ligne pour chacun d'entre eux, l'exemple ci-dessus porte donc sur un seul secondaire. La plupart des colonnes se comprennent aisément.

L'adresse IP du serveur est l'information principale pour distinguer les secondaires s'il y en a plusieurs.

`application_name` peut être fourni par le secondaire dans sa chaîne de connexion `primary_conninfo`. Il est conseillé de le renseigner pour la supervision, (ou bien `cluster_name`).

`state` est à `streaming` quand tout va bien. Quand un secondaire vient de se connecter, `state` affiche `catchup` le temps de revenir au moins une fois à un *lag* nul.

`sync_state` vaut `async` dans le cas d'une réplication asynchrone. Avec une réplication synchrone, ce sera `sync`, `potential` ou `quorum`. Si la connection a échoué, la ligne n'existe simplement pas.

`backend_start` indique l'heure de connexion, et `reply_time` l'heure du dernier message envoyé par le secondaire.

Les quatre LSN permettent de suivre la réception, l'enregistrement et le rejeu sur le secondaire, grâce aux fonctions évoquées plus haut :

```
SELECT application_name,
  pg_size_pretty(pg_wal_lsn_diff( pg_current_wal_lsn(), replay_lsn )) AS retard_rejeu
FROM pg_stat_replication ;
```

```
 application_name | retard_rejeu
------------------+----------------
 secondaire3      | 15 MB
 secondaire2      | 0 bytes
```

Le service **streaming_delta** de la sonde check_pgactivity[1] ne fait pas autrement pour suivre les volumétries à recevoir, à appliquer et à rejouer.

Les différents champs `*_lag` indiquent le retard temporel des secondaires, ce qui est très pratique pour repérer un secondaire en retard ou en pause.

`write_lag` mesure le délai entre l'enregistrement dans les journaux en local et la notification de l'enregistrement dans le cache disque du secondaire (ce délai est important en mode synchrone avec `synchronous_commit` à `remote_write`) mais sans attendre l'écriture physique (`sync`).

`flush_lag` mesure le délai jusqu'à confirmation que les données modifiées soient bien écrites sur disque au niveau du serveur *standby* (ce délai est celui à suivre en mode synchrone avec `synchronous_commit` à `on`).

`replay_lag` mesure le délai jusqu'au rejeu des transactions sur le secondaire, les rendant visibles aux requêtes des utilisateurs (ce délai est à surveiller si `synchronous_commit` est à `remote_apply`)

[1] https://github.com/OPMDG/check_pgactivity

La sortie d'écran plus haut indique que la réception des données sur le secondaire est rapide, mais le rejeu a 3 secondes de retard.

6.2.3 Autres vues pour le streaming

- S'il y un slot
  - `pg_replication_slots`
- Sur le secondaire
  - `pg_stat_wal_receiver`

pg_replication_slots :

Toujours depuis le primaire, pour savoir où en sont les serveurs secondaires, éventuellement déconnectés, utilisant un slot de réplication, consulter aussi la vue `pg_replication_slots` :

```
SELECT * FROM pg_replication_slots ;
```

```
-[ RECORD 1 ]-------+--------------
slot_name           | secondaire1
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | f
active_pid          |
xmin                |
catalog_xmin        |
restart_lsn         | 0/A5F8310
confirmed_flush_lsn |
safe_wal_size       | 5374099280
two_phase           | f
conflicting         |
-[ RECORD 2 ]-------+--------------
slot_name           | secondaire2
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 29287
xmin                |
catalog_xmin        |
restart_lsn         | 0/AEC3B40
```

```
confirmed_flush_lsn |
safe_wal_size       | 5374099280
two_phase           | f
conflicting         |
```

pg_stat_wal_receiver :

Sur le secondaire, on peut consulter aussi la vue `pg_stat_wal_receiver` pour voir la connexion en cours :

```
SELECT * FROM pg_stat_wal_receiver \gx
```

```
-[ RECORD 1 ]---------+------------------------------------------------------------
pid                   | 696088
status                | streaming
receive_start_lsn     | 14/AC000000
receive_start_tli     | 1
written_lsn           | 15/78CB1F8
flushed_lsn           | 15/78CB1F8
received_tli          | 1
last_msg_send_time    | 2023-12-19 12:05:45.275257+01
last_msg_receipt_time | 2023-12-19 12:05:45.275532+01
latest_end_lsn        | 15/78CB1F8
latest_end_time       | 2023-12-19 12:05:45.273271+01
slot_name             | secondaire3
sender_host           | /var/run/postgresql
sender_port           | 5432
conninfo              | user=postgres passfile=/var/lib/postgresql/.pgpass
 ↪  channel_binding=prefer dbname=replication host=/var/run/postgresql port=5432
 ↪  application_name=secondaire3 fallback_application_name=16/secondaire3
 ↪  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
```

Noter que le `primary_conninfo` d'origine est complété de nombreux paramètres par défaut.

6.3 SUPERVISION (LOG SHIPPING)

- Le primaire ne sait rien
- Supervision de l'archivage comme pour du PITR
  - `pg_stat_archiver`
- Secondaire :
  - `pg_wal_lsn_diff()`
  - traces
  - calcul du retard manuel (`pg_last_wal_replay_lsn()`)

Si le secondaire est en *log shipping* (par choix ou parce que le secondaire a trop de retard et a basculé dans ce mode), la supervision est plus compliquée.

Le primaire étant déconnecté du secondaire, `pg_stat_replication` ne contiendra rien sur ce secondaire.

Côté primaire, on vérifiera que l'archivage se fait correctement, notamment avec la vue `pg_stat_archiver`.

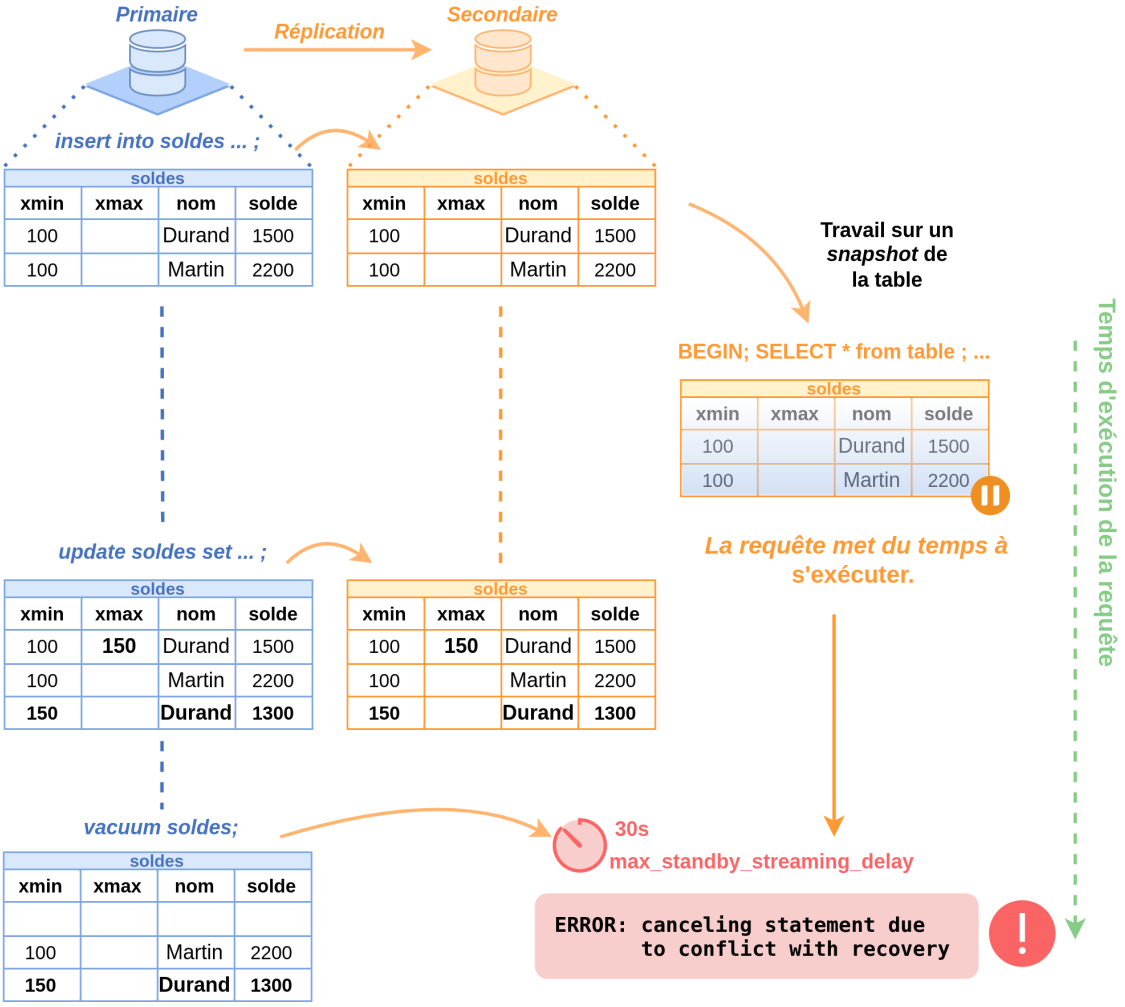
Côté secondaire, les traces permettent de vérifier que les journaux sont récupérés et appliqués, ou de connaître la cause des erreurs : `restore_command` mal paramétrée, problème d'accès aux journaux, etc.

Le calcul du retard ci-dessus reste possible, mais il faudra aller chercher où est le secondaire dans le flux des WAL en y exécutant la fonction `pg_last_wal_replay_lsn()`.

Le service **hot_standby_delta** de la sonde check_pgactivity[2] facilite cela : elle se connecte au primaire et au secondaire, et calcule l'écart, pour lever une alerte au besoin. Il peut être utile de la déployer même sur une instance habituellement en *streaming* pour suivre un rattrapage long.

[2] https://github.com/OPMDG/check_pgactivity

6.4 CONFLITS DE RÉPLICATION



6.4.1 Détection des conflits de réplication

- Une requête en lecture pose des verrous
  - conflit possible avec changements répliqués !
- Vue `pg_stat_database_conflicts` (secondaires)
- Traces :
  - `log_recovery_conflict_waits` (v14+)

Source des conflits :

Sur un primaire, le MVCC garantit qu'une requête ne sera pas gênée si elle lit des lignes dans des blocs qu'une autre requête est en train de modifier.

Mais le primaire ne sait à priori rien des requêtes sur un secondaire. Sur ce dernier, un conflit peut survenir entre l'application des modifications provenant du primaire d'une part, et l'exécution d'une requête (en lecture seule) d'autre part.

Comme les modifications de la réplication doivent s'enregistrer dans l'ordre de leur émission, si une requête bloque l'application d'une modification, elle bloque en fait l'application de toutes les modifications suivantes pour ce serveur secondaire.

Un exemple simple de conflit est l'exécution d'une requête sur une base que la réplication veut supprimer. PostgreSQL attend un peu avant de forcer l'application des modifications. S'il doit forcer, il sera contraint d'annuler les requêtes en cours, voire de déconnecter les utilisateurs. Évidemment, cela ne concerne que les requêtes et/ou les utilisateurs gênants.

Suivi des conflits :

La table `pg_stat_database_conflicts` du catalogue système n'est renseignée que sur les serveurs secondaires d'une réplication. Elle contient le nombre de conflits détectés sur ce secondaire par type de conflit (conflit sur un tablespace, conflit sur un verrou, etc.). Elle contient une ligne par base de données :

```
SELECT * FROM pg_stat_database_conflicts
WHERE datname='postgres' ;
```

```
-[ RECORD 1 ]----+---------
datid            | 12857
datname          | postgres
confl_tablespace | 0
confl_lock       | 0
confl_snapshot   | 3
confl_bufferpin  | 2
confl_deadlock   | 0
```

Le total se retrouve dans `pg_stat_database`.

En version 14 apparaît le paramètre `log_recovery_conflict_waits`. Son activation est conseillée. Il permet de tracer toute attente due à un conflit de réplication. Il n'est donc valable et pris en compte que sur un serveur secondaire.

6.4.2 Prévenir les conflits de réplication

- `wal_standby_streaming_delay`
- `hot_standby_feedback` à `on` + `wal_receiver_status_interval` (10s)
- gênent le vacuum !

Gestion fine des délais pour réduire les conflits :

Avant d'appliquer un journal, ou extrait de journal, qui entre en conflit avec des requêtes en cours sur le secondaire, PostgreSQL attend un certain délai. `max_standby_archive_delay` et `max_standby_streaming_delay` sont des délais d'attente provenant respectivement du *log shipping* ou du *streaming*, Par défaut, c'est 30 secondes. Monter l'un ou l'autre de ces paramètres peut être suffisant si l'on peut tolérer que la réplication soit brièvement bloquée.

Avant la version 16, il existait un paramètre `vacuum_defer_cleanup_age` qui demande au `VACUUM` d'attendre un certain nombre de transactions avant de recycler des lignes mortes. Ce paramètre a été supprimé car le calcul était délicat, et il générait de la fragmentation. Il était désactivé par défaut.

hot_standby_feedback :

De manière plus fine (et plus simple), les serveurs secondaires peuvent envoyer des informations au serveur primaire sur les requêtes en cours d'exécution, pour tenter de prévenir au moins les conflits lors du nettoyage des enregistrements (action effectuée par le `VACUUM`). Il faut pour cela activer le paramètre `hot_standby_feedback` (à `off` par défaut). Le serveur secondaire envoie alors des informations au serveur primaire à une certaine fréquence, configurée par le paramètre `wal_receiver_status_interval`, soit 10 secondes par défaut.

Ces paramètres doivent être maniés avec précaution, car ils peuvent causer une fragmentation des tables sur le serveur primaire, certes pas forcément plus importante que si les requêtes sur le secondaire avaient été lancées sur le primaire. Attention notamment s'il y a un slot de réplication et `hot_standby_feedback` à `on` (voir plus bas) !

Gestion des déconnexions du secondaire

Grâce à cet envoi d'informations, PostgreSQL peut savoir si un serveur secondaire est indisponible, par exemple suite à une coupure réseau ou à un arrêt brutal du serveur secondaire. Rappelons que si jamais le serveur secondaire est indisponible, le primaire coupe la connexion avec le secondaire après un temps déterminé par le paramètre `wal_sender_timeout` (1 minute par défaut), Pour éviter des coupures intempestives, il faut donc conserver `wal_receiver_status_interval` à une valeur inférieure à celle de `wal_sender_timeout`.

6.5 CONTRÔLE DE LA RÉPLICATION

- `pg_wal_replay_pause()` : mettre en pause le **rejeu**
- `pg_wal_replay_resume()` : reprendre
- `pg_is_wal_replay_paused()` : statut
- Utilité :
  - requêtes longues
  - `pg_dump` depuis un secondaire

Lancer un `pg_dump` depuis un serveur secondaire est souvent utile pour ne pas charger le primaire, mais ce n'est pas simple à cause des risques d'annulation de requêtes en cas de conflits. L'exécution d'un `pg_dump` peut durer très longtemps et ce dernier travaille en exécutant des requêtes, parfois très longues (notamment `COPY`) et donc facilement annulées même après configuration des paramètres `max_standby_*_delay`. Il faut donc pouvoir mettre en pause l'application de la réplication avec les fonctions suivantes :

- `pg_wal_replay_pause()`, pour mettre en pause la réplication sur le serveur secondaire où est exécutée cette commande ;
- `pg_wal_replay_resume()`, pour relancer la réplication sur un serveur secondaire où la réplication avait été précédemment mise en pause ;
- `pg_is_wal_replay_paused()`, pour savoir si la réplication est en pause sur le serveur secondaire où est exécutée cette commande.

Ces fonctions s'exécutent uniquement sur les serveurs secondaires et la réplication n'est en pause que sur le serveur secondaire où la fonction est exécutée. Il est donc possible de laisser la réplication en exécution sur certains secondaires et de la mettre en pause sur d'autres.

Plus généralement, cette technique est applicable pour figer des secondaires et y effectuer de très longues requêtes qui n'ont pas besoin des données les plus récentes enregistrées sur le primaire.

Noter qu'il s'agit bien de figer le *rejeu* des journaux, pas leur transmission. Le serveur secondaire ainsi figé stocke les journaux et pourra les réappliquer plus tard. Même une réplication synchrone, dans sa version la moins stricte, reste ainsi possible.

6.6 RÉPLICATION SYNCHRONE

Le primaire attend l'enregistrement sur le secondaire.

- Comment configurer ?
- Comment limiter l'impact sur les performances ?

La réplication synchrone est fréquemment demandée sur tous les moteurs de bases de données.

En réplication **a**synchrone, quand une transaction est validée, le serveur primaire rend la main à l'utilisateur lorsqu'il a fini d'enregistrer les données dans ses journaux de transactions sur disque. Il n'attend donc pas de savoir si le serveur secondaire a reçu les données, et encore moins si elles sont enregistrées sur son disque. Le problème survient quand le serveur primaire s'interrompt soudainement et qu'il faut basculer le serveur secondaire en serveur primaire. Les dernières données enregistrées sur le serveur primaire n'ont peut-être pas eu le temps d'arriver sur le serveur secondaire. Par conséquent, on peut se trouver dans une situation où le serveur indique une transaction comme enregistrée, alors qu'après le *failover* elle n'est plus visible.

Avec une réplication synchrone, le serveur primaire ne valide la transaction auprès de l'utilisateur qu'à partir du moment où le serveur secondaire synchrone a lui aussi reçu/écrit/rejoué la donnée sur disque (selon le mode). Le premier intérêt de la réplication synchrone est donc de s'assurer qu'en cas de *failover*, aucune donnée ne soit perdue. Le second intérêt peut être d'avoir des serveurs secondaires renvoyant exactement la même chose au même moment que le primaire.

L'immense inconvénient de la réplication synchrone est la latence supplémentaire due aux échanges entre les serveurs pour chaque transaction. En effet, il ne faut pas seulement attendre que le serveur primaire fasse l'écriture, il faut aussi attendre l'écriture sur le serveur secondaire sans parler des interactions et des latences réseau. Même si le coût semble minime, il reste cependant présent, et dépend aussi de la qualité du réseau : la durée d'un aller-retour réseau est souvent du même ordre de grandeur (milliseconde) que bien des petites transactions, voire plus élevée. Pour des serveurs réalisant beaucoup d'écritures, le coût n'en sera que plus grand.

Même si le mode peut se choisir transaction par transaction, noter que la réplication d'une transaction synchrone doit attendre la réception, voire le rejeu, de toutes les transactions précédentes. Une grosse transaction asynchrone peut donc ralentir la transmission ou le rejeu de transactions synchrones.

Enfin, la réplication synchrone a un autre danger : si le serveur synchrone ne répond pas, la transaction ne sera pas validée sur le primaire. Du point de vue de l'application, un `COMMIT` ne rendra pas la main. PostgreSQL permet de déclarer plusieurs serveurs synchrones pour réduire le risque.

Ce sera donc du cas par cas. Pour certains, la réplication synchrone sera obligatoire (due à un cahier des charges réclamant aucune perte de données en cas de *failover*). Pour d'autres, malgré l'intérêt de la réplication synchrone, la pénalité à payer sera intolérable. Nous allons voir les différentes options pour limiter les inconvénients.

6.6.1 Secondaires synchrones

- Par défaut : réplication physique **a**synchrone
- Secondaires synchrones :

```
# s1 synchrone, s2 en dépannage
synchronous_standby_names = 'FIRST 1 (s1, s2)'
# 2 synchrones au moins
synchronous_standby_names = 'ANY 2 (s1,s2,s3)'
# n'importe quel secondaire
synchronous_standby_names = '*'
```

- Plusieurs synchrones simultanés possibles
  - ou un quorum

Par défaut, la réplication fonctionne en asynchrone. La mise en place d'un mode synchrone est très simple.

Sur le(s) secondaire(s) synchrone(s) :

Il n'y a rien à configurer de plus. Par contre il est crucial de sécuriser la disponibilité de l'instance.

Sur le primaire :

Comme le paramètre `synchronous_commit` est déjà à `on` par défaut, il ne reste qu'à déclarer les serveurs secondaires synchrones avec le paramètre `synchronous_standby_names`, en séparant par des virgules les différentes instances secondaires synchrones. Il est possible d'indiquer le nombre de serveurs synchrones simultanés. Les serveurs surnuméraires sont des synchrones potentiels.

Pour que `s1` soit un secondaire synchrone, et que `s2` et `s3` le deviennent si `s1` ne répond pas, on a plusieurs syntaxes au choix :

```
synchronous_standby_names = '1 (s1,s2,s3)'
synchronous_standby_names = 'FIRST 1 (s1, s2, s3)'
-- syntaxe à ne plus utiliser
synchronous_standby_names = 's1,s2,s3'
```

Dans l'exemple suivant, `s1` et `s2` seront tous les deux synchrones, `s3` ne le sera pas, sauf défaillance d'un des premiers.

```
synchronous_standby_names = '2 (s1,s2,s3)'
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

`*` remplace la liste des secondaires :

```
# un secondaire désigné synchrone dans la liste, les autres en secours
synchronous_standby_names = '1 (*)'
```

```
synchronous_standby_names = 'FIRST 1(*)'
synchronous_standby_names = '*'
```

Il est possible de se baser sur un quorum. Par exemple, pour que la transaction synchrone s'achève dès que 2 serveurs sur les 3 indiqués l'ont enregistrée, et quels qu'il soient, on écrira :

```
synchronous_standby_names = 'ANY 2 (s1,s2,s3)'
```

Si l'on ne veut pas spécifier les secondaires manuellement, cette syntaxe est très pratique :

```
synchronous_standby_names = 'ANY 2 (*)'
```

Il est parfois nécessaire d'utiliser des guillemets droits :

```
synchronous_standby_names = 'ANY 2 (sec1,"sec-2","sec 3")'
```

La comparaison entre l'`application_name` des connexions de réplication et la liste de serveurs spécifiée dans `synchronous_standby_names` n'est pas sensible à la casse, que l'on utilise des guillemets droits ou non. Il n'y a pas de validation des noms. En cas de faute de frappe, PostgreSQL cherchera donc à être synchrone avec un serveur non connecté, ce qui va bloquer les transactions.

S'il existe des serveurs secondaires non listés dans `synchronous_standby_names`, ils seront implicitement répliqués de manière asynchrone, donc sans impact sur les performances.

Mais comment indiquer le nom d'un serveur secondaire ? Ce nom dépend d'un paramètre de connexion appelé `application_name`, que le client définit librement. Il doit apparaître dans la chaîne de connexion du serveur secondaire au serveur primaire, c'est-à-dire `primary_conninfo`, et différer pour chaque secondaire. Par exemple :

```
primary_conninfo = 'user=user_repli host=prod application_name=s2'
```

Sur le primaire, le nom apparaîtra dans la vue `pg_stat_replication`, champ `application_name`. Ce nom est indépendant de l'éventuel slot de réplication (`primary_slot_name`), même s'ils sont souvent identiques.

6.6.2 Niveau de synchronicité & performances

- Niveau de synchronicité :

```
SET synchronous_commit = off / local / remote_write / on / remote_apply
```

- Ajustable par base/utilisateur/session/transaction
- Risque de blocage du primaire à cause des secondaires !

Pour définir le mode de fonctionnement exact, `synchronous_commit` peut prendre plusieurs valeurs. En ordre croissant de sécurité, ce sont les suivantes :

off :

La transaction est directement validée dans le cache du serveur primaire, mais elle sera être écrite plus tard dans les journaux et sur le disque. Évidement, les secondaires ne sont pas synchrones non plus.

Ce paramétrage peut causer la perte des transactions non encore écrites dans les journaux si le serveur se crashe. La durée d'activité potentiellement perdue est d'au maximum 3 fois la valeur de `wal_writer_delay` (soit au total 0,6 s par défaut). Par contre, il n'y a pas de risque de corruption.

Même sans réplication, `synchronous_commit = off` offre de gros gains de performance dans le cas de nombreuses petites transactions,

C'est à savoir pour tous les cas où la perte des dernières transactions validées ne porte pas à conséquence grave (au milieu d'un batch que l'on relancera par exemple). On réduit en effet l'impact en performance de l'opération de synchronisation sur disque des journaux, sans risquer de corruption de données.

local :

On force le mode asynchrone. La transaction est validée lorsque les données ont été écrites et synchronisées sur le disque de l'instance primaire. En revanche, l'instance primaire ne s'assure pas que le secondaire a reçu la transaction.

S'il n'y a pas de secondaire synchrone, `on` et `local` sont équivalents.

Si le primaire disparaît, il peut y avoir perte de transactions validées et non reçues par un secondaire.

remote_write :

Le primaire synchronise ses journaux, bien sûr, et attend que les journaux soient écrits sur le disque du secondaire via le système d'exploitation, mais sans avoir demandé le vidage du cache système sur disque (`fsync`). Les informations sont donc écrites sur le disque du primaire, mais uniquement dans la mémoire système du secondaire.

Il est donc possible de perdre des données si l'instance secondaire crashe en même temps que le primaire.

`remote_write` impacte beaucoup moins les performances que la valeur `on`, et la fenêtre de perte de données est bien moins importante que le mode asynchrone, mais toujours présente.

L'instance primaire ne s'assure pas non plus que le secondaire a **rejoué** la transaction. Le rejeu des journaux peut effectivement durer un certain temps. Deux requêtes exécutées au même moment sur

le primaire et un secondaire peuvent renvoyer des résultats différents. Ce peut être important dans certains cas.

Le délai que `remote_write` impose se mesure dans `pg_stat_replication`, champ `write_lag`.

on (défaut) :

Sans réplication synchrone, il s'agit du fonctionnement normal, où les journaux de transaction sont synchronisés sur disque (`fsync`) avant que la transaction soit considérée comme validée.

Avec des secondaires synchrones, PostgreSQL attend que l'enregistrement associé au COMMIT soit écrit durablement dans les journaux de transactions des instances primaire **et** secondaire(s). L'impact en performances est donc assez lourd.

Il n'y a donc pas de perte de données en cas de crash (sauf pertes des disques des **deux**, ou plus, ou des machines).

La sécurité étant assurée par l'enregistrement des journaux, le primaire n'attend pas que le secondaire ait réellement rejoué les données pour rendre la main à son client. Le secondaire peut accuser un certain retard (voire avoir mis le rejeu de pause). Là encore, deux requêtes exécutées au même moment sur le primaire et un secondaire peuvent renvoyer des résultats différents.

Le délai que `synchronous_commit` à `on` impose se mesure dans `pg_stat_replication`, champ `flush_lag`.

remote_apply :

C'est le mode de synchronisation le plus poussé. Non seulement les modifications doivent être enregistrées dans les journaux du secondaire, et synchronisées sur son disque, mais le secondaire doit les avoir rejouées pour que PostgreSQL confirme la validation de la transaction au client.

Cette méthode est la seule garantissant qu'une transaction validée sur le serveur primaire sera visible sur le secondaire. Évidemment, elle rajoute encore une latence supplémentaire.

`remote_apply` n'est pas une garantie absolue que les serveurs primaire et secondaires renverront tous la même information au même moment : si un secondaire ne répond pas ou a du retard, la session sera bloquée sur le primaire, et son résultat n'y sera pas encore visible ; mais les secondaires qui fonctionnent bien auront déjà rejoué les données modifiées et les afficheront ! Il n'y a aucune synchronisation entre différents secondaires, et un secondaire ne peut pas savoir que le primaire attend un autre secondaire avant de valider la transaction. Ce problème est rare car une réplication synchrone est à éviter sur une liaison instable.

Le délai que `remote_apply` entraîne se mesure dans `pg_stat_replication`, champ `replay_lag`.

Tableau récapitulatif :

Table 6/ .1: Durée de retour du `COMMIT` de synchronisation des données lors de la validation d'une transaction, et facteur principal de cette durée, selon le paramètre `synchronous_commit`

	Durée	Facteur contraignant (primaire)	Facteur contraignant (secondaire)
off	0	Aucun	Aucun
local	Selon disque	Écriture dans `pg_wal`	Aucun
remote_write	`write_lag`	(idem)	Écriture dans la RAM du secondaire
on	`flush_lag`	(idem)	Écriture dans `pg_wal` du secondaire
remote_apply	`write_lag`	(idem)	Rejeu des données en RAM du secondaire

Les valeurs en `*_lag` sont des champs de `pg_stat_replication`.

Synchronicité différente suivant les cas :

`synchronous_commit` peut être défini dans `postgresql.conf` bien sûr, mais aussi par utilisateur, par base, par utilisateur, par session, voire par transaction :

```
ALTER ROLE batch_user SET synchronous_commit TO off ;
ALTER DATABASE audit  SET synchronous_commit TO local ;
SET synchronous_commit TO on ;        -- dans la session
SET LOCAL synchronous_commit TO remote_apply ;  -- dans la transaction
```

Il est conseillé de n'utiliser la synchronisation que pour les modifications les plus importantes et vitales, et la désactiver pour les cas où la performance en écriture prime, ou si vous pouvez relancer l'opération en cas de crash. À vous de définir la bonne valeur par défaut pour `synchronous_commit`, selon les données, les utilisateurs, les applications, et bien sûr l'impact sur les performances.

Par contre, pour modifier `synchronous_standby_names`, il vous faudra modifier `postgresql.conf` ou passer par `ALTER SYSTEM`, puis recharger la configuration.

En cas de problème :

Il faut savoir qu'en cas d'indisponibilité du ou des secondaire(s) synchrone(s), pour que des transactions synchrones bloquées puissent se terminer, le plus simple est de retirer le secondaire problématique de `synchronous_standby_names` depuis une autre session :

```
SHOW synchronous_standby_names ;

 synchronous_standby_names
---------------------------
 2 (s2,s3)

-- s2 ne répond plus
ALTER SYSTEM SET synchronous_standby_names TO 's3';
SELECT pg_reload_conf();
```

Une alternative est de débrayer le mode synchrone. Cela désactivera aussi le mode synchrone vers d'autres secondaires encore en place.

```
ALTER SYSTEM SET synchronous_commit TO 'local' ;
SELECT pg_reload_conf();
```

Mais les sessions bloquées ne verront pas tout de suite le changement de configuration. Il faudra leur envoyer un signal pour qu'elles se terminent. Elles seront bien validées, dans les journaux du primaire au moins.

```
SELECT pg_cancel_backend(2868749) ;

 pg_cancel_backend
-------------------
 t
```

Apparaît alors le message suivant dans les traces :

```
WARNING:  canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have been
 ↪  replicated to the standby.
```

6.7 RÉPLICATION EN CASCADE

- Un secondaire peut fournir les informations de réplication
- Décharger le serveur primaire de ce travail
- Diminuer la bande passante du serveur primaire

Imaginons un système PostgreSQL installé à Paris et un serveur secondaire installé à Marseille. Il s'avère que le site de Marseille devient plus important et qu'un deuxième serveur secondaire doit y être installé.

Si ce deuxième serveur secondaire se connecte directement sur le primaire à Paris, la consommation de la bande passante va doubler. PostgreSQL permet au deuxième serveur secondaire de se connecter au premier (donc en local dans notre exemple) pour récupérer les informations de réplication. La bande passante est ainsi mieux maîtrisée.

La configuration d'un tel système est très simple. Il suffit d'indiquer l'adresse IP ou l'alias du serveur secondaire à la place de celui du serveur primaire dans le paramètre `primary_conninfo` du fichier `postgresql.conf` du deuxième serveur secondaire.

Si un secondaire est promu et devient primaire, cela n'a pas d'impact sur ses propres secondaires.

6.8 DÉCROCHAGE D'UN SECONDAIRE

- Par défaut, le primaire n'attend **pas** les secondaires pour recycler ses WAL
  - risque de « décrochage » !
- 3 solutions :
- archivage en plus du *streaming*
  - bascule automatique
  - mutualisation avec sauvegarde PITR
- Slot de réplication
- Garder des journaux
  - `wal_keep_size` (v13+) / `wal_keep_segments` (<13)

Par défaut, le primaire n'attend **pas** que le serveur secondaire ait obtenu tous les journaux avant de recycler ses journaux.

Le secondaire peut donc se retrouver à demander au principal des informations que celui-ci n'a même plus à disposition car il a recyclé les journaux concernés. Cela peut arriver si la liaison est mauvaise, instable, ou si le secondaire a peiné à réappliquer les journaux pour une raison ou une autre, voire s'il a été déconnecté un certain temps. Le secondaire ne peut alors plus continuer la réplication : il « décroche » (tout comme après la perte d'un journal en *log shipping*).

Ce phénomène peut intervenir même sur un serveur fraîchement copié, si le maître évolue trop vite.

Il faut reconstruire le secondaire, ce qui est peut être très gênant avec une base volumineuse.

Une réplication synchrone ne protège pas de ce problème, car toutes les transactions ne sont pas forcément synchrones. De plus, l'impact en performance est sévère. `hot_standby_feedback` et `vacuum_defer_cleanup_age` (<= v15) ne protègent pas non plus.

Il existe plusieurs moyens pour éviter le décrochage :

L'archivage comme sécurisation du *streaming*

Une réplication par *log shipping* peut être configurée en plus de la réplication par flux. Comme une sauvegarde PITR du principal est très souvent en place, il ne coûte pas grand-chose de permettre au secondaire d'y puiser les journaux manquants.

Ainsi, si la réplication par *streaming* décroche, le secondaire bascule sur la restauration par *log shipping* et va puiser dans le dépôt d'archives, dont l'historique couvre généralement plusieurs jours, voire semaines. Une fois le retard rattrapé, le secondaire ne trouvera plus de nouveaux journaux et rebasculera sur la réplication par *streaming*, qui fonctionnera à nouveau.

Un inconvénient est qu'il faut bien penser à tester les deux modes de réplication pour ne pas avoir de mauvaise surprise le jour où le *streaming* décroche.

Cette configuration est très fréquente, et même recommandée, surtout avec une sauvegarde PITR déjà en place.

Slots de réplication

Un secondaire peut informer son primaire de là où il en est au moyen d'un « slots de réplication ». Le primaire sait ainsi quels journaux sont encore nécessaires à ses secondaires et ne les recycle pas. Cette technique est également très courante. Nous allons la voir plus bas.

Garder des journaux

La dernière méthode est moins recommandée mais peut être utile : elle consiste à paramétrer `wal_keep_size` sur le primaire, par exemple :

```
wal_keep_size = '16GB'
```

Les journaux de transaction bons à recycler seront en fait conservés temporairement à hauteur de la volumétrie indiquée. Un secondaire en retard a alors plus de chances que le primaire n'ait pas déjà effacé les journaux dont il a besoin.

C'est le moyen le plus simple, mais il gaspille du disque de façon permanente. Surtout, il ne garantit pas d'éviter un décrochage si la quantité à conserver a été sous-estimée.

6.8.1 Sécurisation par log shipping

- `archive_command` / `restore_command`
  - script par l'outil PITR
  - ou `cp`, `scp`, `lftp`, `rsync`, script…
- Nettoyage
  - rétention des journaux si outil PITR
  - ou outil dédié :

```
archive_cleanup_command = '/usr/pgsql-14/bin/pg_archivecleanup -d
↪  rep_archives/ %r'
```

La sécurisation par l'archivage consiste donc à permettre au serveur secondaire de rattraper son retard avant de redémarrer sa connexion de réplication.

Manuellement :

Il suffit qu'`archive_command` et `restore_command` soient correctement configurés et indiquent où copier les archives, et comment les récupérer. La mise en place est la même que lors de la mise en place d'une sauvegarde physique[3]. La `restore_command` est ignorée si le secondaire a rebasculé en *streaming*.

Les serveurs secondaires ont cependant la responsabilité de supprimer les journaux devenus inutiles pour ne pas saturer l'espace disque. Afin d'automatiser ce nettoyage, on définit sur le secondaire le paramètre `archive_cleanup_command`.

La commande qui s'y trouve est appelée périodiquement (même si le *streaming* fonctionne), après chaque *restartpoint* (l'équivalent d'un *checkpoint* sur un secondaire), afin de supprimer les archives devenues inutiles pour le secondaire. Généralement, on se contente d'appeler un outil dédié, nommé `pg_archivecleanup`[4] :

```
archive_cleanup_command = '/usr/pgsql-16/bin/pg_archivecleanup  depot_archives/ %r'
```

La situation se complique si un même dépôt d'archives est partagé par plusieurs secondaires…

Avec un outil de sauvegarde PITR :

La situation est plus simple s'il existe déjà une sauvegarde PITR par un outil comme pgBackRest ou barman[5] : `archive_command` comme `restore_command` sont fournies dans leur documentation.

La purge des journaux étant aussi gérée par cet outil on ne configurera bien sûr pas `archive_cleanup_command` !

6.8.2 Slot de réplication : mise en place

- Slot de réplication sur le primaire :
  - `max_replication_slots`
  - NB : non répliqué !
  - création manuelle :

```
SELECT pg_create_physical_replication_slot ('nomsecondaire') ;
```

- Secondaire :
  - dans `postgresql.conf`

```
primary_slot_name = 'nomsecondaire'
```

  - redémarrage de l'instance (<v13) ou rechargement (v13+)

[3] https://dali.bo/i2_html
[4] https://docs.postgresql.fr/current/pgarchivecleanup.html
[5] https://dali.bo/i4_html

Le paramètre `max_replication_slots` doit être supérieur à 0. Par défaut il vaut 10, ce qui suffit souvent. S'il faut le modifier, un redémarrage est nécessaire.

Un slot de réplication se crée sur le primaire par un appel de fonction et en lui attribuant un nom :

```
SELECT pg_create_physical_replication_slot ('nomsecondaire');
```

Traditionnellement le nom est celui du secondaire qui va l'utiliser. Cela facilite la supervision mais n'a rien d'obligatoire.

Sur le secondaire, on ajoute dans `postgresql.conf` la mention du slot à utiliser :

```
primary_slot_name = 'nomsecondaire'
```

Un slot est propre à l'instance et ne fait pas partie des objets répliqués. Lors d'une restauration PITR ou une bascule, il doit fréquemment être recréé manuellement.

En cas de réplication en cascade, un secondaire peut avoir ses propres slots de réplication dédiés à ses propres secondaires.

6.8.3 Slot de réplication : avantages & risques

- Avantages :
  - plus de risque de décrochage des secondaires
  - supervision facile : `pg_replication_slots`
  - utilisable par pg_basebackup
- Risque : accumulation des journaux
  - danger pour le primaire !
  - sécurité : `max_slot_wal_keep_size` (v13+)
- Risque : vacuum bloqué
  - `hot_standby_feedback` ?

Le slot de réplication garantit au secondaire que son primaire ne recyclera pas les journaux dont il aura encore besoin. Le secondaire peut donc prendre un retard conséquent sans risque de décrochage.

Il est facile de voir à quel point se trouve un secondaire avec la vue `pg_replication_slots` (noter les champs `active` et `restart_lsn`), qui complète `pg_stat_replication` :

```
SELECT * FROM pg_replication_slots ;
```

```
-[ RECORD 1 ]+-----------
slot_name           | s3
plugin              | ø
slot_type           | physical
datoid              | ø
database            | ø
temporary           | f
active              | t
active_pid          | 3951267
xmin                | 3486363
catalog_xmin        | ø
restart_lsn         | 14/ACDD9E10
confirmed_flush_lsn | ø
wal_status          | reserved
safe_wal_size       | 5370962416
two_phase           | f
conflicting         | ø
```

Avec pg_basebackup :

pg_basebackup, déjà évoqué plus haut, utilise les slots pour garantir que sa sauvegarde sera complète. Ses options exactes varient suivant les versions. pg_basebackup est capable de créer ce slot (option `--create-slot`) qui sera conservé ensuite.

Risque d'accumulation des journaux :

Par contre, les slots ont un grave inconvénient : en cas de problème prolongé sur un secondaire, les journaux vont s'accumuler sur le primaire, sans limitation de taille ni de durée, déclenchant une saturation de la partition de `pg_wal` dans le pire des cas — et l'arrêt du primaire.

Certes, la supervision de l'espace disque fait partie des bases de la supervision, mais les journaux s'accumulent parfois très vite lors d'une mise à jour massive.

Il est donc important de détruire tout slot dont le secondaire a été désactivé ou est hors ligne pour un certain temps (quitte à devoir reconstruire le secondaire) :

```
SELECT pg_drop_replication_slot ('nomsecondaire');
```

Les plus prudents se limiteront donc à une réplication par *streaming* classique sans slot, couplée au *log shipping* pour éviter le décrochage. Rappelons que l'archivage peut lui aussi laisser les journaux s'accumuler en cas de problème sur le serveur cible de l'archivage.

À partir de PostgreSQL 13, le paramètre `max_slot_wal_keep_size` permet de limiter la quantité de WAL conservés par les slots de réplication. Au-delà, le primaire ne garantit plus la conservation. Le secondaire risque à nouveau de décrocher, mais une longue indisponibilité ne risque plus de saturer le disque du primaire.

En production, il est conseillé de toujours définir `max_slot_wal_keep_size` (à une valeur élevée au besoin) si l'on crée un slot de réplication. En effet, l'expérience montre que les slots de réplication sont souvent oubliés.

Risque sur le vacuum sur le primaire :

Le slot permet au primaire de mémoriser durablement la transaction où s'est arrêté le secondaire (`pg_replication_slots.xmin` est renseigné), à condition que `hot_standby_feedback` soit à `on`.

Avec un slot de réplication actif et `hot_standby_feedback` à `on`, si le secondaire est durablement déconnecté, non seulement les journaux de transaction vont s'accumuler sur le primaire, mais le vacuum y sera inefficace jusqu'au rétablissement de la réplication de ce secondaire ou la destruction du slot !
Ce problème de vacuum persiste même si l'on a paramétré `max_slot_wal_keep_size` pour éviter la saturation des journaux…

Selon l'utilisation, on peut donc préférer monter `max_standby_streaming_delay` plutôt que de laisser `hot_standby_feedback` à `on`.

6.9 SYNTHÈSE DES PARAMÈTRES

6.9.1 Serveur primaire

Log shipping	Streaming
`wal_level = replica` *	`wal_level = replica` *
`archive_mode = on` *	
`archive_command` *	
`archive_library`	
`archive_timeout`	`wal_sender_timeout`
	`max_wal_senders`
	`max_replication_slots`
	`wal_keep_size`
	`max_slot_wal_keep_size` *

(*) paramètres indispensables, généralement modifiés par rapport à l'installation par défaut, ou d'utilisation fortement conseillés

Ne figurent pas les paramètres disparus dans les toutes dernières versions, généralement inutilisés auparavant.

6.9.2 Serveur secondaire

Log shipping	Streaming
`wal_level = replica` *	`wal_level = replica` *
`restore_command` *	
`archive_cleanup_command`	
(selon outil)	`primary_conninfo` *
	`wal_receiver_timeout`
	`hot_standby`
	`primary_slot_name` *
`max_standby_archive_delay`	`max_standby_streaming_delay`
	`hot_standby_feedback`
	`wal_receiver_status_interval`

(*) paramètres indispensables, généralement modifiés par rapport à l'installation par défaut, ou d'utilisation fortement conseillés

Ne figurent pas les paramètres disparus dans les toutes dernières versions, généralement inutilisés auparavant.

6.10 CONCLUSION

- Système de réplication fiable…
  - et très complet

PostgreSQL possède de nombreuses fonctionnalités de réplication très avancées, telle que le choix du synchronisme de la réplication à la transaction près, ce qui en fait un système aujourd'hui très complet.

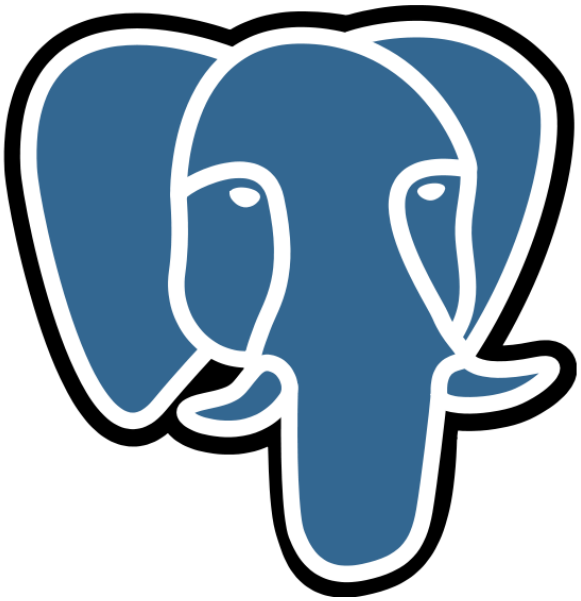
6.10.1 Questions

N'hésitez pas, c'est le moment !

6.11 QUIZ

https://dali.bo/w2b_quiz

7/ Les outils de réplication



7.1 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.

7.1.1 Au menu

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

7.2 (RE)CONSTRUIRE UN SECONDAIRE

- pg_basebackup
- rsync
- outils PITR
- pg_rewind

7.2.1 (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.

7.2.2 (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 !

7.2.3 (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.

7.2.4 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.

7.3 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.

7.3.1 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`…).

7.3.2 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_*`…).

7.4 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.

7.4.1 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

7.4.2 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.

7.4.3 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.

7.4.4 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/

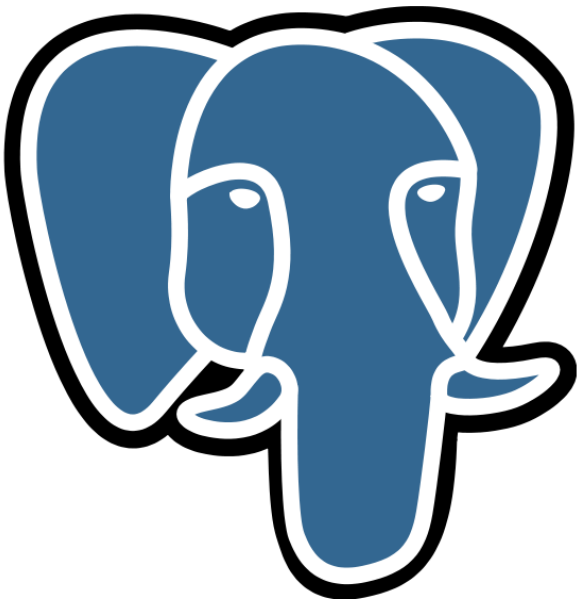
7.5 CONCLUSION

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

7.5.1 Questions

N'hésitez pas, c'est le moment !

8/ Réplication logique



8.1 OBJECTIFS

- Réplication logique native
  - connaître les avantages et limites
  - savoir la mettre en place
  - savoir l'administrer et la superviser
- Connaître d'autres outils de réplication logique

Nous verrons ici les principes derrière la réplication logique, les différences avec la réplication physique classique, sa mise en place, son administration et sa supervision.

Historiquement sont apparus des outils de réplication logique externes à PostgreSQL, qui peuvent encore rendre des services.

8.1.1 Au menu

- Principes
- Mise en place
- Exemple
- Administration
- Supervision
- Migration majeure avec la réplication logique
- Limitations
- Autres outils de réplication logique

8.2 PRINCIPES DE LA RÉPLICATION LOGIQUE NATIVE

- Réplication logique
  - résout certaines des limitations de la réplication physique
  - native depuis la version 10
  - avant v10 : solutions externes
  - préférer tout de même PostgreSQL >= 14

La réplication physique, qui existe dans PostgreSQL depuis la version 9.0, fonctionne par application de bloc d'octets ou de delta de bloc. Elle a beaucoup évolué mais possède quelques limitations difficilement contournables directement.

La réplication logique apporte des réponses à ces limitations depuis PostgreSQL 10. Seules des solutions tierces apportaient ce type de réplication à PostgreSQL auparavant. Il est préférable d'utiliser une version récente de PostgreSQL (14 au moins) pour profiter des nombreuses améliorations et optimisations.

8.2.1 Réplication physique vs. logique

Physique	Logique
Instance complète	Tables aux choix
Par bloc	Par ligne/colonnes
Asymétrique (1 principal)	Asymétrique / croisée
Depuis primaire ou secondaire	Depuis primaire ou secondaire (v16)
Toutes opérations	Opération au choix
Réplica identique	Destination modifiable
Même architecture	-
Mêmes versions majeures	-
Synchrone/Asynchrone	Synchrone/Asynchrone

Principe & limites de la réplication physique :

La réplication physique est une réplication au niveau bloc. Le serveur primaire envoie au secondaire les octets à ajouter/remplacer dans des fichiers. Le serveur secondaire n'a aucune information sur les objets logiques (tables, index, vues matérialisées, bases de données). Il n'y a donc pas de granularité

possible, c'est forcément l'instance complète qui est répliquée. Cette réplication est par défaut en asynchrone mais il est possible de la configurer en synchrone suivant différents modes.

Malgré ses nombreux avantages, la réplication physique souffre de quelques défauts.

Il est impossible de ne répliquer que certaines bases ou que certaines tables (pour ne pas répliquer des tables de travail par exemple). Il est aussi impossible de créer des index spécifiques ou même des tables de travail, y compris temporaires, sur les serveurs secondaires, vu qu'ils sont strictement en lecture seule.

Un serveur secondaire ne peut se connecter qu'à un serveur primaire de même version majeure. On ne peut donc pas se servir de la réplication physique pour mettre à jour la version majeure du serveur.

La réplication physique peut se faire depuis un serveur secondaire (réplication en cascade).

Enfin, il n'est pas possible de faire de la réplication entre des serveurs d'architectures matérielles ou logicielles différentes (32/64 bits, *little/big endian*, version de bibliothèque C, etc.).

Réplication logique :

La réplication logique propose une solution à tous ces problèmes.

La réplication logique est une réplication du contenu des tables. Plus précisément, elle réplique les résultats des ordres SQL exécutés sur la table publiée et l'applique sur la table cible. Les lignes insérées, modifiées et/supprimées sur le serveur d'origine sont répliquées sur la destination. La table cible peut être modifiée (index notamment), et son contenu différer de la table source.

Elle se paramètre donc table par table, et même opération par opération.

Elle est asymétrique dans le sens où il existe une seule origine des écritures pour une table. Il est possible de réaliser des réplications croisées où un ensemble de tables est répliqué du serveur 1 vers le serveur 2 et un autre ensemble de tables est répliqué du serveur 2 vers le serveur 1, ce qui est une forme limitée de multimaître. En version 16, la possibilité de filtrer l'origine d'une table permet d'éviter les boucles et d'alimenter la même table depuis deux instances différentes. Cette fonctionnalité est encore jeune et peut mener à des problèmes d'intégrité des données (voir notamment cet article de Brian Pace[1]).

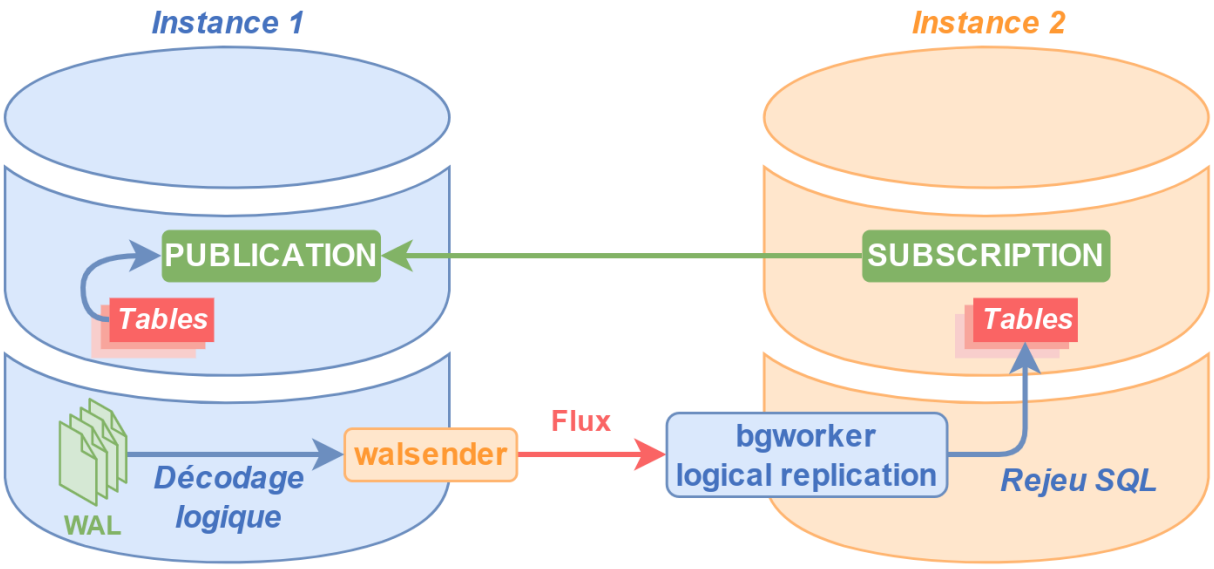
Comme la réplication physique, la réplication logique peut fonctionner en asynchrone ou en synchrone, si l'on accepte l'impact sur les performances.

La réplication logique ne peut se faire depuis un serveur secondaire (en réplication physique) que si l'origine et la destination sont au moins en PostgreSQL 16.

La réplication logique permet de répliquer entre deux serveurs PostgreSQL de versions différentes, et ainsi de procéder à des migrations majeures.

[1] https://www.crunchydata.com/blog/active-active-postgres-16#dont-get-too-carried-away

8.2.2 Schéma de principe de la réplication logique



8.2.3 Quelques termes essentiels

- Serveur origine (publieur/éditeur)
  - publication
- Serveur(s) abonné(s) (*subscriber*)
  - abonnement (*subscription*)

Dans le cadre de la réplication logique, on ne réplique pas une instance vers une autre. On publie les modifications effectuées sur le contenu d'une table à partir d'un serveur. Ce serveur est le serveur origine, ou publieur (*publisher*). Sur ce serveur, on crée des « publications ».

Une publication enregistre un jeu de modifications que d'autres serveurs pourront récupérer en s'abonnant (*subscription*).

De ceci, il découle que :

- le serveur origine est le serveur où les écritures sur une table sont enregistrées pour publication vers d'autres serveurs ;
- les serveurs intéressés par ces enregistrements sont les serveurs destinations ;
- un serveur origine doit proposer une publication des modifications ;
- les serveurs destinations intéressés doivent s'abonner à une publication.

Dans un cluster de réplication, un serveur peut avoir un rôle de serveur origine ou de serveur destination. Il peut aussi avoir les deux rôles. Dans ce cas, il sera origine pour certaines tables et destinations pour d'autres. Il ne peut pas être à la fois origine et destination pour la même table.

NB : dans le texte qui suit, peuvent être utilisés indifféremment les termes publieur/éditeur/origine d'une part, et abonné/souscripteur/destination et abonnement/souscription d'autre part, éventuellement en anglais.

8.2.4 Réplication logique et streaming

La réplication logique utilise le *streaming* :

- `wal_level` = `logical`
- Processus `wal sender`
  - mais pas de `wal receiver`
  - un `logical replication worker` à la place
- Décodage logique des journaux
- Asynchrone / synchrone
- Slots de réplication

La réplication logique utilise le même canal d'informations que la réplication physique : les enregistrements des journaux de transactions. Le transfert se fait par une connexion en *streaming* (ce n'est pas possible en *log shipping*). Pour que les journaux disposent de suffisamment d'informations, le paramètre `wal_level` doit être configuré avec la valeur `logical`.

Une fois cette configuration effectuée et PostgreSQL redémarré sur le serveur origine, le serveur destination peut se connecter au serveur origine dans le cadre de la réplication. Lorsque cette connexion est faite, un processus `wal sender` apparaît sur le serveur origine. Ce processus est en communication avec un processus `logical replication worker` sur le serveur destination.

Comme la réplication physique, la réplication logique peut être configurée en asynchrone comme en synchrone, suivant le même paramétrage (`synchronous_commit`, `synchronous_standby_names`).

Chaque abonné maintient un slot de réplication sur l'instance de l'éditeur. Par défaut, il est créé et supprimé automatiquement avec la souscription. La copie initiale des données crée également des slots de réplication temporaires.

Contrairement à ce qui se passe en réplication physique, l'intégralité des journaux n'est pas transmise. Le `walsender` procède à un « décodage logique » des journaux, que la documentation définit ainsi : Le décodage logique correspond au processus d'extraction de tous les changements persistants sur des tables d'une base de données dans un format cohérent et simple à comprendre, qui peut être interprété sans une connaissance détaillée de l'état interne de la base de données.

Dans PostgreSQL, le décodage logique est implémenté en décodant le contenu des journaux de transaction (WAL), qui décrivent les changements au niveau stockage, dans un format spécifique tel qu'un flux de lignes ou des ordres SQL.

Le décodage logique permet de n'envoyer aux abonnés que les informations qui concernent la table qu'ils ont demandée. Cela permet aussi de n'envoyer que les transactions validées lors du `COMMIT` (du moins dans les cas simples). Cette conversion est hélas un peu gourmande en processeur.

En face, sur une instance PostgreSQL avec des souscriptions, le rôle du `logical replication worker` est de retransmettre les modifications reçues dans les tables concernées.

Noter que le décodage logique permet d'écrire assez facilement des plugins de sortie alternatifs sans outil supplémentaire sur le serveur. Par exemple pg_wal2json[2] permet de s'abonner à une table et de restituer ses modifications sous forme d'objets JSON.

8.2.5 Granularité de la réplication logique

- Par table
  - toutes les tables d'une base
  - toutes les tables d'un schéma (v15+)
  - quelques tables spécifiques
- Granularité d'une table
  - table complète
  - même partitionnée (v13+)
  - uniquement certaines lignes/colonnes (v15+)
- Par opération
  - `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`

La granularité de la réplication physique est simple : c'est l'intégralité de l'instance ou rien.

À l'inverse, la réplication logique propose une granularité à la table près, voire même un niveau en dessous. Une publication se crée en indiquant la table pour laquelle on souhaite publier les modifications. On peut en indiquer plusieurs. On peut en ajouter après en modifiant la publication. Cependant, une nouvelle table ne sera pas ajoutée automatiquement à la publication, sauf dans deux cas précis : la publication a été créée en demandant la publication de toutes les tables (clause `FOR ALL TABLES`) ou, à partir de la version 15, la publication a été créée en demandant la publication de toutes les tables d'un schéma (clause `FOR TABLES IN SCHEMA`).

[2] https://github.com/eulerto/wal2json

À partir de la version 13, il est possible d'ajouter une table partitionnée à une publication. Cette fonctionnalité permet de laisser à PostgreSQL le soin d'ajouter et maintenir à jour la liste des partitions dans la publication. Il est également possible de faire en sorte que les modifications soient publiées avec le nom de la partition finale ou celui de la table partitionnée. Cela permet plus de flexibilité en autorisant de répliquer des données entre des tables avec des structures hétérogènes (partitionnées ou non). Dans le cas d'une réplication depuis une table partitionnée vers une autre table partitionnée, l'agencement des partitions peut être différent.

À partir de la version 15, la granularité est encore plus basse : il est possible de ne filtrer que certaines colonnes et que certaines lignes.

La granularité peut aussi se voir au niveau des opérations de modification réalisées. On peut très bien ne publier que les opérations d'insertion, de modification ou de suppression. Par défaut, tout est publié.

8.2.6 Possibilités sur les tables répliquées

- Possibilités
  - index supplémentaires
  - modification des valeurs
  - colonnes supplémentaires
  - triggers également activables sur la table répliquée
- **Attention à la cohérence des modèles**
- **Attention à ne pas bloquer la réplication logique !**
  - aller au plus simple

La réplication logique permet plusieurs choses impensables en réplication physique. Les cas d'utilisation sont en fait très différents.

On peut rajouter ou supprimer des index sur la table répliquée.

Il est possible de modifier des valeurs dans la table répliquée. Ces modifications sont susceptibles d'être écrasées par des modifications de la table source sur les mêmes lignes. Il est aussi possible de perdre la synchronisation entre les tables, notamment si on modifie la clé primaire.

Les triggers ne se déclenchent par défaut que sur la base d'origine. On peut activer ainsi un trigger sur la table répliquée :

```
ALTER TABLE matable ENABLE REPLICA TRIGGER nom_trigger ;
```

Tout cela est parfois très pratique mais peut poser de sérieux problème de cohérence de données entre les deux instances si l'on ne fait pas attention. On vérifiera régulièrement les erreurs dans les

traces.

Il est dangereux d'ajouter sur la destination des contraintes qui n'existent pas sur les tables d'origine ! Elles ne sont pas forcément contrôlées à l'arrivée (clés étrangères, vérification par triggers…) Et si elles le sont, elles risquent de bloquer la réplication logique. De même, sur la destination, ajouter ou modifier des lignes soumises à des contraintes d'unicité peut empêcher l'insertion de lignes provenant de la source.

En cas de blocage, à cause d'une colonne absente, d'un doublon, d'une autre contrainte sur la cible ou pour une autre raison, il faut corriger sur la destination, puis laisser le stock de données bloquées s'insérer avant de pouvoir faire autre chose. L'alternative est de désactiver ou reconstruire la réplication, ce qui peut poser des problèmes de réconciliation de données.

Il existe quelques cas surprenants. Par exemple, une colonne remplie grâce à une valeur `DEFAULT` sur l'origine sera répliquée à l'identique sur la destination ; mais une colonne calculée (clause `GENERATED` avec expression) sera calculée sur l'origine et sur la destination, éventuellement différemment.

Il est possible de créer une publication sur une table elle-même répliquée. La sécurité pour éviter des boucles n'a été ajoutée qu'avec PostgreSQL 16[3].

Pour que la réplication logique fonctionne sans souci, il faut viser au plus simple, avec un modèle de données sur la destination aussi proche que possible de la source, soigneusement maintenu à jour à l'identique. Éviter de modifier les données répliquées. Au plus, se contenter d'ajouter sur la destination des index non uniques ou des colonnes calculées.
Prévoir dès le début le cas où cette réplication devra être arrêtée et reprise de zéro.

[3] https://amitkapila16.blogspot.com/2023/09/evolution-of-logical-replication.html

8.2.7 Limitations de la réplication logique

- Pas de réplication des requêtes DDL
  - à refaire manuellement
  - être rigoureux et surveiller les traces !
- Pas de réplication des valeurs des séquences
- Pas de réplication des LO (table système)
- Problèmes avec les tables partitionnées (< v13)
- PK/UK conseillée pour les `UPDATE` / `DELETE`
- Coût CPU, disque, RAM
- Réplication déclenchée uniquement lors du `COMMIT` (< v14)
- Attention en cas de bascule/restauration !

La réplication logique n'a pas que des atouts, elle a aussi ses propres limitations.

La première, et plus importante, est qu'elle ne réplique que les changements de données des tables (commandes DML), et pas de la définition des objets de la base de données (commandes DDL). Une exception a été faite à partir de la version 11 pour répliquer les ordres `TRUNCATE` car, même s'il s'agit d'un ordre DDL d'après le standard, cet ordre modifie les données d'une table.

L'ajout (ou la suppression) d'une colonne ne sera pas répliqué, causant de ce fait un problème de réplication quand l'utilisateur y ajoutera des données. La mise à jour sera bloquée jusqu'à ce que les tables abonnées soient aussi mises à jour. Pour éviter le blocage, il est préférable de commencer une opération d'ajout de colonne sur l'abonné, et une opération de suppression de colonne sur le publieur.

D'autres opérations moins évidentes peuvent aussi poser problème, comme une contrainte ou un index supprimé sur l'origine mais pas la cible ; ou un index fonctionnel dont la fonction n'est corrigée que sur la source.

Il faut être rigoureux et surveiller les erreurs dans les traces.

Une table nouvellement créée ne sera pas non plus automatiquement répliquée.

Les tables partitionnées, sur la source ou l'origine, ne sont bien gérées qu'à partir de PostgreSQL 13.

Il n'y a pas de réplication des valeurs des séquences. Les valeurs des séquences sur les serveurs de destination seront donc obsolètes.

Les Large Objects (pour le stockage de gros binaires) étant stockés dans une table système, ils ne sont pas pris en compte par la réplication logique. Il est préférable de passer par le type `bytea` pour les

données binaires.

Il faut que PostgreSQL sache opérer une correspondance entre les lignes des deux instances pour gérer correctement les opérations `UPDATE` et `DELETE` Pour cela, **il est chaudement conseillé qu'il y ait une clé primaire sur chaque table répliquée.** Sinon, il faut définir une clause `REPLICA IDENTITY` sur la table origine. Utiliser un index unique peut convenir :

```
ALTER TABLE nomtable REPLICA IDENTITY USING INDEX nomtable_col_idx ;
```

Si vraiment on n'a pas le choix, on peut définir que l'ensemble des champs de la ligne servira à la correspondance :

```
ALTER TABLE nomtable REPLICA IDENTITY FULL ;
```

Faute d'index, les mises à jour effectuent des *Seq Scan* sur la table de destination, ce qui généralement catastrophique pour les performances. Toutefois, avec PostgreSQL 16, un simple index B-tree ou hash sur la table destination peut permettre d'éviter ces *Seq Scan*[4].

La réplication logique a un coût en CPU (sur les deux instances concernées) relativement important : attention aux petites configurations. Il y a également un coût en RAM et disque (voir plus bas).

La réplication n'est par défaut déclenchée que lors du `COMMIT` sur le primaire, nous verrons que cela peut être optimisé à partir de PostgreSQL 14.

La situation peut devenir compliquée lors d'une restauration ou bascule d'un des serveurs impliqués (voir plus bas).

[4] https://amitkapila16.blogspot.com/2023/09/evolution-of-logical-replication.html

8.3 MISE EN PLACE

Étapes :

- Configuration du serveur origine
- Configuration du serveur destination
- Création d'une publication
- Ajout d'une souscription

Nous allons voir les étapes de configuration dans le cas simple d'une publication origine alimentant un abonnement destination.

8.3.1 Configurer le serveur origine : utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION ;
GRANT SELECT ON ALL TABLES IN SCHEMA monschema TO logrepli ;

# pg_hba.conf
host base_publication  logrepli XXX.XXX.XXX.XXX/XX scram-sha-256
```

Dans le cadre de la réplication avec PostgreSQL, c'est toujours le serveur destination qui se connecte au serveur origine. Pour la réplication physique, on utilise plutôt les termes de serveur primaire et de serveur secondaire mais c'est toujours du secondaire vers le primaire, de l'abonné vers l'éditeur.

Tout comme pour la réplication physique, il est nécessaire de disposer d'un utilisateur PostgreSQL capable de se connecter au serveur origine et capable d'initier une connexion de réplication. Voici donc la requête pour créer ce rôle :

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

Cet utilisateur doit pouvoir lire le contenu des tables répliquées. Il lui faut donc le droit `SELECT` sur ces objets, souvent simplement ceci :

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli;
```

Enfin, la connexion du serveur destination doit être possible sur le serveur origine. Il est donc nécessaire d'avoir une ligne du style :

```
host base_publication   logrepli XXX.XXX.XXX.XXX/XX scram-sha-256
```

en remplaçant `XXX.XXX.XXX.XXX/XX` par l'adresse CIDR du serveur destination. La méthode d'authentification peut aussi être changée suivant la politique interne. Suivant la méthode

d'authentification, il sera nécessaire ou pas de configurer un mot de passe pour cet utilisateur. Ne pas oublier de recharger la configuration.

8.3.2 Configurer le serveur origine : postgresql.conf

- `wal_level = logical`
  - rédémarrage
- `logical_decoding_work_mem = 64MB` (v13+)
  - en-deça : RAM jusque `COMMIT`
  - puis disque
  - ou transmission immédiate (v14+)

Les journaux de transactions doivent disposer de suffisamment d'informations pour que le `wal sender` puisse envoyer les bonnes informations au `logical replication worker`. Le fichier `postgresql.conf` doit donc contenir :

```
wal_level = logical
```

Le défaut est `replica`, et il faudra donc sans doute redémarrer l'instance.

Le paramètre `logical_decoding_work_mem` contrôle la quantité de mémoire allouée à un processus `walsender` pour conserver les modifications en mémoire avant de les stocker sur le disque (et parfois de les envoyer tout de suite au client, voir plus bas).

En effet, la réplication logique, contrairement à la réplication physique, n'est déclenchée que lors d'un `COMMIT` (voir cet article[5]). Par défaut, il n'y a pas d'envoi des données tant que la transaction est en cours, ce qui peut ajouter beaucoup de délai de réplication pour les transactions longues.

`logical_decoding_work_mem` vaut par défaut 64 Mo. Il peut donc être réduit pour baisser l'utilisation de la mémoire des `walsender`, ou augmenté pour réduire les écritures sur le disque.

Avant PostgreSQL 13 et l'apparition de ce paramètre, les modifications d'une transaction étaient stockées en mémoire jusqu'à ce que la transaction soit validée par un `COMMIT`. En conséquence, si cette transaction possédait de nombreuses sous-transactions, chaque `walsender` pouvait allouer énormément de mémoire, menant parfois à un dépassement de mémoire.

[5] http://amitkapila16.blogspot.com/2021/07/logical-replication-of-in-progress.html

8.3.3 Configuration du serveur destination

- Création, si nécessaire, des tables répliquées

```
pg_dump -h origine -s -t la_table la_base | psql la_base
```

Sur le serveur destination, il n'y a pas de configuration à réaliser dans les fichiers `postgresql.conf` et `pg_hba.conf`.

Ensuite, il faut récupérer la définition des objets répliqués pour les créer sur le serveur de destination. Un moyen simple est d'utiliser `pg_dump` et d'envoyer le résultat directement à `psql` pour restaurer immédiatement les objets. Cela se fait ainsi :

```
pg_dump -h origine --schema-only base | psql base
```

Il est aussi possible de sauvegarder la définition d'une seule table en ajoutant l'option `-t` suivi du nom de la table pour avoir son script.

Il est conseillé de déclarer l'objet sur la destination avec la même définition que sur l'origine, mais ce n'est pas obligatoire tant que les mises à jour arrivent à se faire. Les index, notamment, peuvent différer, des types être plus laxistes, des colonnes supplémentaires ajoutées.

8.3.4 Créer une publication

```
CREATE PUBLICATION pub_t1     FOR TABLE t1 ;
CREATE PUBLICATION pub_t1part FOR TABLE t1 (c1, c3);  -- v15
CREATE PUBLICATION pub_tout   FOR ALL TABLES ;
CREATE PUBLICATION pub_public FOR TABLES IN SCHEMA public ; -- v15

CREATE PUBLICATION pub_filtree
FOR TABLE employes  WHERE ( ville = 'Brest' ) ; --v15

… WITH ( publish = 'update, delete, insert, truncate')  -- défaut

… WITH (publish_via_partition_root = false)  -- défaut, v13
```

Une fois que les tables sont définies des deux côtés (origine et destination), il faut créer une publication sur le serveur origine. Cette publication indiquera à PostgreSQL les tables répliquées et les opérations concernées.

La clause `FOR ALL TABLES` permet de répliquer toutes les tables de la base, sans avoir à les nommer spécifiquement. De plus, toute nouvelle table sera répliquée automatiquement dès sa création.

À partir de la version 15, la clause `FOR TABLES IN SCHEMA` permet de répliquer toutes les tables du schéma indiqué sans avoir à nommer les tables spécifiquement. De plus, toute nouvelle table de ce schéma sera répliquée automatiquement dès sa création. (Il faudra tout de même rafraîchir l'abonnement sur le destinataire).

Si on ne souhaite répliquer qu'un sous-ensemble, il faut spécifier toutes les tables à répliquer en utilisant la clause `FOR TABLE` et en séparant les noms des tables par des virgules.

Depuis la version 15, il est possible de ne répliquer que certaines colonnes d'une table, par exemple ainsi : exemple :

```
CREATE PUBLICATION pub1
  FOR TABLE t1 (c1, c3);
```

Toujours depuis cette version, il est possible de ne répliquer que les lignes validant une certaine expression. Par exemple :

```
CREATE PUBLICATION pub_brest
  FOR TABLE employes WHERE (ville='Brest');
```

Par défaut, une table est répliquée intégralement, donc toutes les colonnes et toutes les lignes.

La clause `FOR TABLES` n'est pas obligatoire, la publication peut être vide au départ.

Cette publication est concernée par défaut par toutes les opérations d'écriture (`INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`). Cependant, il est possible de préciser les opérations si on ne les souhaite pas toutes. Pour cela, il faut utiliser le paramètre de publication `publish` en utilisant les valeurs `insert`, `update`, `delete` et/ou `truncate` et en les séparant par des virgules si on en indique plusieurs.

Lorsque l'on publie les modifications sur une table partitionnée, PostgreSQL utilise par défaut le nom de la partition finale. Il est possible de lui demander d'utiliser le nom de la table partitionnée grâce à l'option `publish_via_partition_root = true`. Cela permet de répliquer d'une table partitionnée vers une table normale ou une table partitionnée avec un agencement de partitions différent.

8.3.5 Souscrire à une publication

```
CREATE SUBSCRIPTION nom
    CONNECTION 'infos_connexion'
    PUBLICATION nom_publication [, ...]
    [ WITH ( parametre_souscription [= value] [, ... ] ) ]
```

- `infos_connexion` : chaîne de connexion habituelle
- Par : superutilisateur ou `pg_create_subscription`

Une fois la publication créée, le serveur destination doit s'y abonner. Il doit pour cela indiquer sur quel serveur se connecter et à quelle publication souscrire.

Chaîne de connexion :

Le serveur s'indique avec la chaîne `infos_connexion`, dont la syntaxe est la syntaxe habituelle des chaînes de connexion avec `host`, `port`, `user`, `password`, `dbname`, etc.

Droits :

Pour créer ou modifier la souscription, il faut être superutilisateur, ou posséder le rôle `pg_create_subscription` (à partir de PostgreSQL 16). Dans ce dernier cas, il faut aussi le droit `CREATE` sur la base.

Il y a une subtilité pour le mot de passe : il doit être présent dans la chaîne de connexion (`password = motdepasse`), ce qui impose une méthode de connexion avec mot de passe (par exemple `scram-sha-256` mais pas `peer` ou `ldap`). Le superutilisateur n'a pas cette contrainte.

Une alternative est de créer la souscription en tant que superutilisateur en désactivant le mot de passe avant de changer le propriétaire :

```
ALTER SUBSCRIPTION abonnement SET (password_required = false) ;
ALTER SUBSCRIPTION abonnement OWNER TO erpadmin ;
```

Sans cela, PostgreSQL n'empêchera pas de transférer la propriété d'une souscription à un non-superutilisateur ; mais l'obligation du mot de passe risque de poser divers problèmes lors des modifications de la souscription.

Bien sûr, les accès aux tables répliquées ne nécessite aucun droit sur les souscriptions même.

Autres paramètres :

Le champ `nom_publication` doit être remplacé par le nom de la publication créée précédemment sur le serveur origine.

Les paramètres de souscription sont détaillés ci-dessous.

8.3.6 Options de la souscription (1/2)

Par défaut :

- `connect = true`
  - connexion immédiate
- `copy_data = true`
  - copie initiale des données
- `create_slot = true`
  - création du slot de réplication
- `enabled = true`
  - activation immédiate de la souscription
- `slot_name = <nom de la souscription>`
  - nom du slot de réplication

Les options de souscription sont assez nombreuses et permettent de créer une souscription pour des cas particuliers. (Pour les détails, voir la documentation officielle[6].)

Par exemple, si le serveur destination possède déjà les données du serveur origine, il faut placer le paramètre `copy_data` à la valeur `false` dans la clause `WITH` de `CREATE SUBSCRIPTION`.

`enabled = false` permet de mettre en place la souscription sans la démarrer.

[6] https://docs.postgresql.fr/current/sql-createsubscription.html

8.3.7 Options de la souscription (2/2)

Par défaut :

- `streaming = off`
  - `true` pour envoyer les modifications avant `COMMIT` (v14+)
  - évite de gros fichiers sur le primaire
  - `parallel` : plusieurs workers (v16+)
- `binary = off` (v14+)
  - pour envoyer les données sous un format binaire
- `disable_on_error = false`
  - désactivation de la souscription en cas d'erreurs détectées
- `synchronous_commit = off`
  - surcharge `synchronous_commit` pour les `wal sender`

streaming :

Ce paramètre est très important pour les performances.

Par défaut, le `walsender` de l'origine attend le `COMMIT`, et aussi d'avoir décodé toute la transaction, avant de l'envoyer aux abonnés. De grosses transactions peuvent alors entraîner de la consommation mémoire (jusque `logical_decoding_work_mem`), puis l'apparition d'énormes fichiers temporaires dans le répertoire `pg_replslot` du serveur d'origine.

Depuis la version 14, le client peut demander l'envoi des données au fil de l'eau, sans attendre le `COMMIT`, dès que `logical_decoding_work_mem` est atteint. Le serveur destination stocke alors lui-même les données dans un fichier temporaire, et ne les rejoue qu'à réception du `COMMIT`.

Cette fonctionnalité doit s'activer explicitement avec le paramètre `streaming` au niveau de la souscription depuis le client :

```
CREATE SUBSCRIPTION sub_stream
  CONNECTION 'connection string'
  PUBLICATION pub WITH (streaming = on);
```

ou :

```
ALTER SUBSCRIPTION sub_stream SET (streaming = on);
```

Depuis PostgreSQL 16, on peut même paralléliser :

```
ALTER SUBSCRIPTION sub_stream SET (streaming = parallel);
```

Dans ce cas plusieurs workers apparaissent pour appliquer les modifications en parallèle. S'il n'est pas possible de créer les workers car il y a déjà trop de *background workers*, on revient en pratique au fonctionnement de `streaming = on`.)

binary

Activer le mode binaire est potentiellement plus rapide mais dépend beaucoup des types employés qui ne peuvent pas tous être convertis.

8.4 MISE EN PLACE : EXEMPLE

- Réplication complète d'une base
- Réplication partielle d'une base
- Réplication croisée

Pour rendre la mise en place plus concrète, voici trois exemples de mise en place de la réplication logique. On commence par une réplication complète d'une base, qui permettrait notamment de faire une montée de version. On continue avec une réplication partielle, ne prenant en compte que 2 des 3 tables de la base. Et on finit par une réplication croisée sur la table partitionnée.

8.4.1 Serveurs et schéma

- 4 serveurs
  - **s1**, 192.168.10.1 : origine de toutes les réplications, et destination de la réplication croisée
  - **s2**, 192.168.10.2 : destination de la réplication complète
  - **s3**, 192.168.10.3 : destination de la réplication partielle
  - **s4**, 192.168.10.4 : origine et destination de la réplication croisée
- Schéma
  - 2 tables ordinaires
  - 1 table partitionnée, avec trois partitions

Voici le schéma de la base d'exemple, `b1` :

```
CREATE TABLE t1 (id_t1 serial, label_t1 text);
CREATE TABLE t2 (id_t2 serial, label_t2 text);

CREATE TABLE t3 (id_t3 serial, label_t3 text, clepartition_t3 integer)
  PARTITION BY LIST (clepartition_t3);

CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES IN (1);
CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES IN (2);
CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES IN (3);

INSERT INTO t1 SELECT i, 't1, ligne '||i FROM generate_series(1, 100) i;
```

```
INSERT INTO t2 SELECT i, 't2, ligne '||i FROM generate_series(1, 1000) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 1 FROM generate_series(  1, 100) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 2 FROM generate_series(101, 300) i;
INSERT INTO t3 SELECT i, 't3, ligne '||i, 3 FROM generate_series(301, 600) i;

ALTER TABLE t1 ADD PRIMARY KEY(id_t1);
ALTER TABLE t2 ADD PRIMARY KEY(id_t2);
ALTER TABLE t3 ADD PRIMARY KEY(id_t3, clepartition_t3);
```

8.4.2 Réplication complète

- Configuration du serveur origine
- Configuration du serveur destination
- Création de la publication
- Ajout de la souscription

Pour ce premier exemple, nous allons détailler les quatre étapes nécessaires.

8.4.3 Configuration du serveur origine (1/2)

- Création et configuration de l'utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO logrepli;
```

- Fichier `postgresql.conf`

```
wal_level = logical
```

La configuration du serveur d'origine commence par la création du rôle de réplication. On lui donne ensuite les droits sur toutes les tables. Ici, la commande ne s'occupe que des tables du schéma `public`, étant donné que nous n'avons que ce schéma. Dans le cas où la base dispose d'autres schémas, il serait nécessaire d'ajouter les ordres SQL pour ces schémas.

Les fichiers `postgresql.conf` et `pg_hba.conf` sont modifiés pour y ajouter la configuration nécessaire.

8.4.4 Configuration du serveur origine (2/2)

- Fichier `pg_hba.conf`

```
host b1 logrepli 192.168.10.0/24 trust
```

- Redémarrer le serveur origine
- Attention, dans la vraie vie, ne pas utiliser `trust`
  - et utiliser le fichier `.pgpass`

Comme dit précédemment, les fichiers `postgresql.conf` et `pg_hba.conf` sont modifiés pour y ajouter la configuration nécessaire. Le serveur PostgreSQL du serveur d'origine est alors redémarré pour qu'il prenne en compte cette nouvelle configuration.

Il est important de répéter que la méthode d'authentification `trust` ne devrait jamais être utilisée en production. Elle n'est utilisée ici que pour se faciliter la vie.

8.4.5 Configuration des 4 serveurs destinations

- Création de l'utilisateur de réplication

```
CREATE ROLE logrepli LOGIN REPLICATION;
```

- Création des tables répliquées (sans contenu)

```
createdb -h s2 b1
pg_dump -h s1 -s b1 | psql -h s2 b1
```

Pour cet exemple, nous ne devrions configurer que le serveur **s2** mais tant qu'à y être, autant le faire pour les quatre serveurs destinations.

La configuration consiste en la création de l'utilisateur de réplication. Puis, nous utilisons `pg_dump` pour récupérer la définition de tous les objets grâce à l'option `-s` (ou `--schema-only`). Ces ordres SQL sont passés à `psql` pour qu'il les intègre dans la base **b1** du serveur **s2**.

8.4.6 Créer une publication complète

- Création d'une publication de toutes les tables de la base **b1** sur le serveur origine **s1**

```
CREATE PUBLICATION publi_complete
  FOR ALL TABLES;
```

On utilise la clause `ALL TABLES` pour une réplication complète d'une base.

8.4.7 Souscrire à la publication

- Souscrire sur **s2** à la publication de **s1**

```
CREATE SUBSCRIPTION subscr_complete
  CONNECTION 'host=192.168.10.1 user=logrepli dbname=b1'
  PUBLICATION publi_complete;
```

- Un slot de réplication est créé sur l'origine
- Les données initiales sont immédiatement transférées

Maintenant que le serveur **s1** est capable de publier les informations de réplication, le serveur intéressé doit s'y abonner. Lors de la création de la souscription, il doit préciser comment se connecter au serveur origine et le nom de la publication.

La création de la souscription ajoute immédiatement un slot de réplication sur le serveur origine.

Par défaut, les données initiales de la table `t1` sont immédiatement envoyées du serveur **s1** vers le serveur **s2**.

8.4.8 Tests de la réplication complète

- Insertion, modification, suppression sur les différentes tables de **s1**
- Vérifications sur **s2**
  - toutes doivent avoir les mêmes données entre **s1** et **s2**

Toute opération d'écriture sur la table `t1` du serveur **s1** doit être répliquée sur le serveur **s2**.

Sur le serveur **s1** :

```
INSERT INTO t1 VALUES (101, 't1, ligne 101');
UPDATE t1 SET label_t1=upper(label_t1) WHERE id_t1=10;
DELETE FROM t1 WHERE id_t1=11;
SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
```

```
 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
(2 rows)
```

Sur le serveur **s2** :

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

```
 count
-------
   100
```

```
SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);
```

```
 id_t1 |   label_t1
-------+---------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
(2 rows)
```

8.4.9 Réplication partielle

- Identique à la réplication complète, à une exception…
- Créer la publication partielle

```
CREATE PUBLICATION publi_partielle
  FOR TABLE t1,t2 ;
```

- Souscrire sur **s3** à cette nouvelle publication de **s1**

```
CREATE SUBSCRIPTION subscr_partielle
  CONNECTION 'host=192.168.10.1 user=logrepli dbname=b1'
  PUBLICATION publi_partielle;
```

La mise en place d'une réplication partielle est identique à la mise en place d'une réplication complète à une exception près : la publication doit mentionner la liste des tables à répliquer. Chaque nom de table est séparé par une virgule.

Mise en place :

Cela donne donc dans notre exemple :

```
CREATE PUBLICATION publi_partielle
  FOR TABLE t1,t2;
```

Il ne reste plus qu'à souscrire à cette publication à partir du serveur **s3** avec la requête indiquée.

Vérification :

Sur **s3**, nous n'avons que les données des deux tables répliquées :

```
SELECT count(*) FROM t1;

 count
-------
   100

SELECT count(*) FROM t2;

 count
-------
  1000

SELECT count(*) FROM t3;

 count
-------
     0
```

À noter que nous avons déjà les données précédemment modifiées :

```
SELECT * FROM t1 WHERE id_t1 IN (101, 10, 11);

 id_t1 |   label_t1
-------+--------------
   101 | t1, ligne 101
    10 | T1, LIGNE 10
```

Maintenant, ajoutons une ligne dans chaque table de **s1** :

```
INSERT INTO t1 VALUES (102, 't1, ligne 102');
INSERT INTO t2 VALUES (1001, 't2, ligne 1002');
INSERT INTO t3 VALUES (-1, 't3, cle 1, ligne -1', 1);
```

Et vérifions qu'elles apparaissent bien sur **s3** pour `t1` et `t2`, mais pas pour `t3` :

```
SELECT * FROM t1 WHERE id_t1=102;

 id_t1 |   label_t1
-------+--------------
   102 | t1, ligne 102

SELECT * FROM t2 WHERE id_t2=1001;

 id_t2 |    label_t2
-------+---------------
  1001 | t2, ligne 1002

SELECT * FROM t3 WHERE id_t3 < 0;

 id_t3 | label_t3 | clepartition_t3
-------+----------+-----------------
(0 rows)
```

8.4.10 Réplication croisée

- Écrire sur une table sur **s1**
  - et répliquer sur **s4**
- Écrire sur une (autre) table sur **s4**
  - et répliquer sur **s1**
- Pour compliquer :
  - on utilisera la table partitionnée

La réplication logique ne permet pas pour l'instant de faire du multi-maîtres pour une même table. Cependant, il est tout à fait possible de croiser les réplications, c'est-à-dire de répliquer un ensemble

de tables de serveur **s1** (origine) vers **s4** (destination), de répliquer un autre ensemble en sens inverse, du serveur **s4** vers **s1**.

Pour rendre cela encore plus intéressant, nous allons utiliser la table `t3` et ses partitions. Le but est de pouvoir écrire dans la partition `t3_1` sur **s1** et dans la partition `t3_2` sur **s4**, simulant ainsi une table où il sera possible d'écrire sur les deux serveurs à condition de respecter la clé de partitionnement.

Pour le mettre en place, nous allons travailler en deux temps :

- nous allons commencer par mettre en réplication `t3_1` ;
- et nous finirons en mettant en réplication `t3_2`.

8.4.11 Réplication de t3_1 de s1 vers s4

- Créer la publication partielle sur **s1**

```
CREATE PUBLICATION publi_t3_1
  FOR TABLE t3_1;
```

- Y souscrire sur **s4**

```
CREATE SUBSCRIPTION subscr_t3_1
  CONNECTION 'host=192.168.10.1 user=logrepli dbname=b1'
  PUBLICATION publi_t3_1;
```

- Configurer **s4** comme serveur origine
  - `wal_level`, `pg_hba.conf`

Rien de bien nouveau ici, il s'agit d'une réplication partielle. On commence par créer la publication sur le serveur **s1** et on souscrit à cette publication sur le serveur **s4**.

Cependant, le serveur **s4** n'est plus seulement un serveur destination, il devient aussi un serveur origine. Il est donc nécessaire de le configurer pour ce nouveau rôle. Cela passe par une configuration similaire et symétrique à celle vue pour **s1** :

- Fichier `postgresql.conf` :

```
wal_level = logical
```

(Si ce n'était pas déjà fait, il faudra redémarrer l'instance PostgreSQL sur **s4**).

- Fichier `pg_hba.conf` :

```
host all logrepli 192.168.10.0/24 trust
```

(Ne pas oublier de recharger la configuration.)

8.4.12 Réplication de t3_2 de s4 vers s1

- Créer la publication partielle sur **s4**

```
CREATE PUBLICATION publi_t3_2
  FOR TABLE t3_2;
```

- Y souscrire sur **s1**

```
CREATE SUBSCRIPTION subscr_t3_2
  CONNECTION 'host=192.168.10.4 user=logrepli dbname=b1'
  PUBLICATION publi_t3_2;
```

Là-aussi, rien de bien nouveau. On crée la publication sur le serveur **s4** et on souscrit à cette publication sur le serveur **s1**.

8.4.13 Tests de la réplication croisée

- Insertion, modification, suppression sur `t3` (partition 1) sur **s1**
  - Vérifications sur **s4** : les nouvelles données doivent être présentes
- Insertion, modification, suppression sur `t3` (partition 2) sur **s4**
  - Vérifications sur **s1** : les nouvelles données doivent être présentes

Sur **s1** :

```
SELECT * FROM t3 WHERE id_t3 > 999;

 id_t3 | label_t3 | clepartition_t3
-------+----------+-----------------
(0 rows)

INSERT INTO t3 VALUES (1001, 't3, ligne 1001', 1);
SELECT * FROM t3 WHERE id_t3>999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
```

Sur **s4** :

```
SELECT * FROM t3 WHERE id_t3 > 999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
```

```
INSERT INTO t3 VALUES (1002, 't3, ligne 1002', 2);
SELECT * FROM t3 WHERE id_t3 > 999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
  1002 | t3, ligne 1002 |               2
```

Sur **s1** :

```
SELECT * FROM t3 WHERE id_t3>999;
```

```
 id_t3 |    label_t3    | clepartition_t3
-------+----------------+-----------------
  1001 | t3, ligne 1001 |               1
  1002 | t3, ligne 1002 |               2
(2 rows)
```

8.5 ADMINISTRATION

- Processus
- Fichiers
- Procédures
  - Empêcher les écritures sur un serveur destination
  - Que faire pour les DDL ?
  - Gérer les opérations de maintenance
  - Gérer les sauvegardes

Dans cette partie, nous allons tout d'abord voir les changements de la réplication logique au niveau du système d'exploitation, et tout particulièrement au niveau des processus et des fichiers.

Ensuite, nous regarderons quelques procédures importantes d'administration et de maintenance.

8.5.1 Processus

- Serveur origine
  - `wal sender`
- Serveur destination
  - `logical replication launcher`
  - `logical replication worker`

Tout comme il existe un processus `wal sender` communiquant avec un processus `wal receiver` dans le cadre de la réplication physique, il y a aussi deux processus discutant ensemble dans le cadre de la réplication logique.

Le `logical replication launcher` est toujours exécuté. Ce processus a pour but de demander le lancement d'un `logical replication apply worker` lors de la création d'une souscription. Ce worker se connecte au serveur origine et applique toutes les modifications dont **s1** lui fait part. Si la connexion se passe bien, un processus `wal sender` est ajouté sur le serveur origine pour communiquer avec le *worker* sur le serveur destination.

Sur notre serveur **s2**, destinataire pour la publication complète du serveur **s1**, nous avons les processus suivant :

```
postmaster -D /opt/postgresql/datas/s2
postgres: checkpointer process
postgres: writer process
postgres: wal writer process
postgres: autovacuum launcher process
postgres: bgworker: logical replication launcher
postgres: bgworker: logical replication apply worker for subscription 16445
```

Le serveur **s1** est origine de trois publications (d'où les 3 `wal sender`) et destinataire d'une souscription (d'où le seul `logical replication apply worker`). Il a donc les processus suivants :

```
postmaster -D /opt/postgresql/datas/s1
postgres: checkpointer process
postgres: writer process
postgres: wal writer process
postgres: autovacuum launcher process
postgres: bgworker: logical replication launcher
postgres: bgworker: logical replication apply worker for subscription 16573
postgres: wal sender process logrepli [local] idle
postgres: wal sender process logrepli [local] idle
postgres: wal sender process logrepli [local] idle
```

8.5.2 Synthèse des paramètres sur le serveur origine

Paramètre	Valeur
`wal_level`	`logical`
`logical_decoding_work_mem`	`64MB` ou plus
`max_slot_wal_keep_size`	0 (à ajuster)
`wal_sender_timeout`	1 min
`max_wal_senders`	10 (parfois à ajuster)
`max_replication_slots`	10 (parfois à ajuster

À part les deux premiers, ces paramètres ont la même utilité que pour une réplication physique. Les valeurs par défaut sont généralement suffisantes, mais doivent parfois être augmentées.

Exception : `max_slot_wal_keep_size` doit être mis en place à une valeur élevée pour qu'un slot de réplication très en retard ou oublié ne sature le répertoire `pg_wal` du serveur origine.

8.5.3 Synthèse des paramètres sur le serveur destination

Paramètre	Valeur
`max_worker_processes`	8 (parfois à ajuster)
`max_logical_replication_workers`	4 (parfois à ajuster)

`max_logical_replication_workers` spécifie le nombre maximal de *workers* de réplication logique (*leader*, *parallel*, de synchronisation). Ils sont pris dans la réserve définie par `max_worker_processes`.

Si les valeurs sont trop basses, les réplications seront bloquées. Il faudra augmenter ces valeurs et redémarrer le serveur.

8.5.4 Fichiers (serveur origine)

- 2 répertoires importants
- `pg_replslot`
  - slots de réplication
  - 1 répertoire par slot (+ slots temporaires)
  - 1 fichier `state` dans le répertoire
  - fichiers `.snap` (volumétrie !)
- `pg_logical`
  - métadonnées
  - snapshots

La réplication logique maintient des données dans deux répertoires : `pg_replslot` et `pg_logical`.

`pg_replslot` contient un répertoire par slot de réplication physique ou logique. On y trouvera aussi des slots temporaires lors de l'initialisation de la réplication logique.

`pg_replslot` contient aussi les snapshots des transactions en cours (fichiers `.snap`). Il peut donc atteindre une taille importante si le serveur exécute beaucoup de transactions longues avec du volume en écriture, ou si l'abonné met du temps à répliquer les données. Il est donc important de surveiller la place prise par ce répertoire.

`pg_logical` contient des métadonnées et une volumétrie beaucoup plus faible.

À cela s'ajoutent les journaux de transaction conservés dans `pg_wal/` en fonction de l'avancement des slots de réplication.

8.5.5 Empêcher les écritures sur un serveur destination

- Par défaut, toutes les écritures sont autorisées sur le serveur destination
  - y compris écrire dans une table répliquée avec un autre serveur comme origine
- Problèmes
  - serveurs non synchronisés
  - blocage de la réplication en cas de conflit sur la clé primaire
- Solution
  - révoquer le droit d'écriture sur le serveur destination
  - mais ne pas révoquer ce droit pour le rôle de réplication !

Sur **s2**, nous allons créer un utilisateur applicatif en lui donnant tous les droits sur les tables répliquées, entre autres :

```
CREATE ROLE u1 LOGIN;
GRANT ALL ON ALL TABLES IN SCHEMA public TO u1;
```

Maintenant, nous nous connectons avec cet utilisateur et vérifions s'il peut écrire dans la table répliquée :

```
\c b1 u1
INSERT INTO t1 VALUES (103, 't1 sur s2, ligne 103');
```

C'est bien le cas, contrairement à ce que l'on aurait pu croire instinctivement. Le seul moyen d'empêcher ce comportement par défaut est de lui supprimer les droits d'écriture :

```
\c b1 postgres
REVOKE INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public FROM u1;
\c b1 u1
INSERT INTO t1 VALUES (104);

ERROR:  permission denied for relation t1
```

L'utilisateur **u1** ne peut plus écrire dans les tables répliquées.

Sans cette interdiction, on peut arriver à des problèmes très gênants. Par exemple, nous avons inséré dans la table `t1` de **s2** la valeur 103 :

```
SELECT * FROM t1 WHERE id_t1=103;

 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s2, ligne 103
```

Cette ligne n'apparaît pas sur **s1** :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 | label_t1
-------+----------
(0 rows)
```

De ce fait, on peut l'insérer sur la table `t1` de **s1** :

```
INSERT INTO t1 VALUES (103, 't1 sur s1, ligne 103');
```

Et maintenant, on se trouve avec deux serveurs désynchronisés :

- sur **s1** :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s1, ligne 103
(1 row)
```

- sur **s2** :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s2, ligne 103
(1 row)
```

Notez que le contenu de la colonne `label_t1` n'est pas identique sur les deux serveurs.

Ce n'est pas le seul problème : cette valeur insérée sur **s1** va devoir être répliquée. Le processus de réplication logique n'arrive alors plus à appliquer les données sur **s2**, avec ces messages dans les traces :

```
LOG:  logical replication apply worker for subscription "subscr_complete" has started
ERROR:  duplicate key value violates unique constraint "t1_pkey"
DETAIL:  Key (id_t1)=(103) already exists.
LOG:  worker process: logical replication worker for subscription 16445 (PID 31113)
 ↪  exited with exit code 1
```

Il faut corriger manuellement la situation, par exemple en supprimant la ligne de `t1` sur le serveur **s2** :

```
DELETE FROM t1 WHERE id_t1=103;
```

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 | label_t1
-------+----------
(0 rows)
```

Au bout d'un certain temps, le *worker* est relancé, et la nouvelle ligne est finalement disponible :

```
SELECT * FROM t1 WHERE id_t1=103;
```

```
 id_t1 |       label_t1
-------+----------------------
   103 | t1 sur s1, ligne 103
(1 row)
```

Dans des cas plus complexes et avec plus de données, la réconciliation des données peut devenir très complexe et chronophage.

8.5.6 Que faire pour les DDL ?

- Les opérations DDL ne sont pas répliquées
- De nouveaux objets ?
  - les déclarer sur tous les serveurs du cluster de réplication
  - tout du moins, ceux intéressés par ces objets
- Changement de définition des objets ?
  - à réaliser sur chaque serveur

Seules les opérations DML sont répliquées pour les tables ciblées par une publication. Toutes les opérations DDL sont ignorées, que ce soit l'ajout, la modification ou la suppression d'un objet, y compris si cet objet fait partie d'une publication.

Il est donc important que toute modification de schéma soit effectuée sur toutes les instances d'un cluster de réplication. Ce n'est cependant pas requis. Il est tout à fait possible d'ajouter un index sur un serveur sans vouloir l'ajouter sur d'autres. C'est d'ailleurs une des raisons de passer à la réplication logique.

Par contre, dans le cas du changement de définition d'une table répliquée (ajout ou suppression d'une colonne, par exemple), il est nettement préférable de réaliser cette opération sur tous les serveurs intégrés dans cette réplication.

8.5.7 Que faire pour les nouvelles tables ?

- Créer la table sur origine **et** destination
- Publication `FOR ALL TABLES` / `FOR TABLES IN SCHEMA`
  - prise en compte automatique ajouter la table aux souscriptions concernées :

```
-- origine
ALTER PUBLICATION … ADD TABLE …, TABLE … ;
ALTER SUBSCRIPTION … REFRESH PUBLICATION ;
```

La création d'une table est une opération DDL. Elle est donc ignorée dans le contexte de la réplication logique. Si l'on veut la répliquer, il faut d'abord créer la table manuellement dans la base destinataire. Puis, plusieurs cas se présentent :

- si la publication a été définie `FOR ALL TABLES`, la nouvelle table sera prise en compte immédiatement ;
- si la publication a été définie `FOR ALL TABLES IN SCHEMA`, et que la nouvelle table est dans le bon schéma, elle sera aussi prise en compte ;
- si la publication liste les tables une à une, il va falloir l'ajouter manuellement à la publication ainsi :

```
ALTER PUBLICATION … ADD TABLE … ;
```

Dans les deux cas, sur les serveurs destinataires, il va falloir rafraîchir les souscriptions :

```
ALTER SUBSCRIPTION … REFRESH PUBLICATION ;
```

Si l'on a oublié de créer la table sur le destinataire, cela provoquera une erreur :

```
ERROR:  relation "public.t4" does not exist
```

Si la publication contient des tables partitionnées, la même commande doit être exécutée lorsque l'on ajoute ou retire des partitions à une de ces tables partitionnées.

Il est possible d'ajouter une table à une publication définie sur un schéma différent avec `FOR ALL TABLES IN SCHEMA`.

Exemple :

Sur le serveur origine **s1**, on crée la table `t4`, on lui donne les bons droits, et on insère des données :

```
CREATE TABLE t4 (id_t4 integer, PRIMARY KEY (id_t4));
GRANT SELECT ON TABLE t4 TO logrepli;
INSERT INTO t4 VALUES (1);
-- optionnel pour les publications table à table
ALTER PUBLICATION publi_partielle ADD TABLE t4 ;
```

Sur le serveur **s2**, on regarde le contenu de la table `t4` :

```
SELECT * FROM t4;

ERROR:  relation "t4" does not exist
LINE 1: SELECT * FROM t4;
                      ^
```

La table n'existe pas. En effet, la réplication logique ne s'occupe que des modifications de contenu des tables, pas des changements de définition. Il est donc nécessaire de créer la table sur le serveur destination, ici **s2** :

```
CREATE TABLE t4 (id_t4 integer, primary key (id_t4));
SELECT * FROM t4;

 id_t4
-------
(0 rows)
```

Elle ne contient toujours rien. Ceci est dû au fait que la souscription n'a pas connaissance de la réplication de cette nouvelle table. Il faut donc rafraîchir les informations de souscription :

```
ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION;

SELECT * FROM t4;

 id_t4
-------
     1
```

8.5.8 Comment ajouter une nouvelle colonne ?

- 1. Ajouter la colonne sur l'abonné
- 2. Puis ajouter la colonne sur le publieur
- Si le contraire : pas grave, la réplication reprendra une fois les colonnes ajoutées

Un publieur qui a une table avec plus de colonnes qu'un abonné posera problème à la première insertion de ligne ou modification de la colonne, et ce message apparaîtra dans les traces :

```
ERROR:  logical replication target relation "public.t4" is missing replicated
↪  column: "c9"
```

Le contraire n'est pas vrai : un abonné peut avoir une table ayant plus de colonnes que la même table sur le publieur. C'est un des intérêt de la réplication logique. Les colonnes n'ont pas non plus besoin d'être dans le même ordre sur les deux instances.

Il est donc conseillé d'ajouter la nouvelle colonne sur l'abonné en premier lieu, puis de faire la même opération sur le publieur.

Si jamais vous faites l'opération dans le sens inverse et qu'une ligne est insérée avant avoir terminé l'opération, la réplication sera en erreur jusqu'à ce que l'opération soit terminée.

8.5.9 Comment supprimer une colonne ?

- 1. Supprimer la colonne sur le publieur
- 2. Supprimer la colonne sur l'abonné
- Si le contraire : pas grave, la réplication reprendra une fois les colonnes supprimées

Comme indiqué ci-dessus, une table peut avoir plus de colonnes sur l'abonné mais pas sur le publieur. De ce fait, pour supprimer une colonne, il convient de commencer par la supprimer sur le publieur, puis de la supprimer sur l'abonné.

Si jamais vous faites l'opération dans le sens inverse et qu'une ligne est insérée avant la fin de l'opération, la réplication sera en erreur jusqu'à ce que l'opération soit terminée.

8.5.10 Comment ajouter une nouvelle contrainte ?

- 1. Ajouter la contrainte sur le publieur
- 2. Ajouter la contrainte sur l'abonné
- Si incohérence : bloquage de la réplication

L'ajout d'une contrainte identique sur les deux machines doit se faire d'abord sur le primaire. Sans cela, il y a une fenêtre pour que de nouvelles données violant cette contrainte soient insérées dans l'origine et bloquent la réplication.

Ajoutons que les contraintes ne sont pas obligatoirement les mêmes sur l'origine et la destination. Pour faciliter l'administration, c'est tout de même conseillé. Rappelons qu'une clé primaire ou unique est nécessaire pour repérer plus efficacement les lignes.

En cas de différence, il vaut donc mieux que les contraintes les plus strictes soient posées sur le publieur. Une ligne insérée sans problème sur l'origine et violant une contrainte sur la destination bloquera la réplication.

8.5.11 Comment corriger une erreur de réplication ?

- Si les données diffèrent entre les serveurs, il faut corriger manuellement les données
- Si blocage
  - publication arrêtée
  - pas de recyclage des journaux → accumulation → danger !
- Puis, avancer le pointeur du slot de réplication
  - fonction `pg_replication_slot_advance()`
  - outil `pg_waldump` ou extension `pg_walinspect`

Voici un exemple complet de correction d'une erreur de réplication.

Commençons par mettre en place une réplication logique entre deux serveurs **s1** (port 5432) et **s2** (port 5433). Cette réplication prend en compte la seule table de la base tests1.

Dans la base **tests1** sur le publieur (**s1**) :

```
CREATE TABLE t1(c1 integer, c2 integer);
ALTER TABLE t1 ADD PRIMARY KEY(c1);
CREATE PUBLICATION pub1 FOR ALL TABLES;
```

Dans la base **tests1** sur l'abonné (**s2**) :

```
CREATE TABLE t1(c1 integer, c2 integer);
ALTER TABLE t1 ADD PRIMARY KEY(c1);
CREATE SUBSCRIPTION sub1 CONNECTION 'port=5432 dbname=tests1' PUBLICATION pub1;
```

À partir de maintenant, toute écriture sur **s1** sera lisible aussi sur **s2** :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (1,1), (2,2);
INSERT 0 2
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
```

```
(2 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
(2 rows)
```

Ajoutons maintenant une contrainte sur l'abonné :

```
tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres".
tests1=# ALTER TABLE t1 ADD CHECK (c2<10);
ALTER TABLE
```

Tout ajout se passera bien, sur **s1** et **s2**, tant que la contrainte est respectée :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (3,3), (4,4);
INSERT 0 2
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

Par contre, si la contrainte n'est pas respectée, l'ajout se fera uniquement sur **s1** (qui n'a pas la contrainte) :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (11,11);
INSERT 0 1
tests1=# TABLE t1;
```

```
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
 11 | 11
(5 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

Les traces du serveur **s2** nous expliquent pourquoi :

```
LOG:  logical replication apply worker for subscription "sub1" has started
ERROR:  new row for relation "t1" violates check constraint "t1_c2_check"
DETAIL:  Failing row contains (11, 11).
CONTEXT:  processing remote data for replication origin "pg_16390" during message
 ↪  type "INSERT" for replication target relation "public.t1" in transaction 748,
 ↪  finished at 0/1C442C8
LOG:  background worker "logical replication worker" (PID 194674) exited with exit
 ↪  code 1
```

Ce message sera répété tant que l'erreur ne sera pas corrigée.

De plus, aucune autre donnée de réplication ne passera par ce slot de réplication tant que l'erreur n'est pas corrigée.

Par exemple, ces cinq lignes sont bien insérées dans la table `t1` du serveur **s1** mais pas dans celle du serveur **s2** :

```
postgres=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres".
tests1=# INSERT INTO t1 (c1) SELECT generate_series(5, 9);
INSERT 0 5
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
 11 | 11
```

```
  5 |
  6 |
  7 |
  8 |
  9 |
(10 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

Ceci est très problématique car les journaux de transactions ne pourront pas être recyclés sur le serveur **s1** tant que le problème n'est pas réglé. Pour éviter une perte du service sur **s1**, il est donc essentiel de corriger le problème le plus rapidement possible.

Supprimer la contrainte résoudra facilement et rapidement le problème… si la contrainte n'avait pas lieu d'être. Si, au contraire, cette contrainte est nécessaire, et si nous avons seulement oublié de l'ajouter sur le serveur **s1**, il faut pouvoir supprimer la ligne 11 sur **s1**, ajouter la contrainte sur **s1** et reprendre la réplication sur **s2**.

Voyons comment faire cela. La suppression de la ligne 11 est simple. Profitons-en en plus pour récupérer l'identifiant de transaction qui a créé la ligne :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# DELETE FROM t1 WHERE c1=11 RETURNING xmin;
 xmin
------
  748
(1 row)

DELETE 1
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
```

```
  8 |
  9 |
(9 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
(4 rows)
```

La ligne 11 est bien supprimée du serveur **s1** mais ça n'a pas débloqué pour autant la situation sur le serveur **s2**. C'est normal. L'information d'ajout de la ligne est dans les journaux disponibles sur le serveur **s2**. Supprimer la ligne sur **s1** ne supprime pas l'enregistrement de l'insertion préalable de cette ligne des journaux de transactions.

En attendant, ajoutons la contrainte sur **s1** pour ne plus avoir de « mauvaises » données insérées puis répliquées :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres".
tests1=# ALTER TABLE t1 ADD CHECK (c2<10);
ALTER TABLE
```

Rétablissons maintenant la réplication sur le serveur **s2**. Nous ne voulons pas appliquer l'enregistrement qui insère la ligne 11 sur le serveur **s2**. (Sinon il serait possible de lever temporairement la contrainte, et laisser la ligne 11 s'insérer ; puis le `DELETE` ci-dessus s'appliquerait, et on pourrait remettre la contrainte en place sur **s2**.)

Pour cela, il faut pouvoir avancer le pointeur du prochain enregistrement à rejouer pour notre slot de réplication. Il existe une fonction dédiée : `pg_replication_slot_advance()`. Cette fonction prend en premier argument le nom du slot de réplication à modifier, et en deuxième argument la nouvelle position dans les journaux de transactions. Il va donc falloir trouver l'emplacement suivant dans la transaction qui a insérée cette ligne 11.

Nous savons à quel emplacement le slot est bloqué grâce à la vue `pg_replication_slots` et son champ `confirmed_flush_lsn`, qui indique le dernier enregistrement reçu, mais pas forcément appliqué :

```
tests1=# \c tests1 - - 5432
tests1=# SELECT confirmed_flush_lsn FROM pg_replication_slots WHERE slot_name='sub1';
 confirmed_flush_lsn
---------------------
 0/1C44248
(1 row)
```

Maintenant, il faut décoder les enregistrements après cet emplacement là. Auparavant, il fallait utiliser l'outil `pg_walinspect`. Voici ce que nous donne la fonction `pg_get_wal_records_info()` de cette

extension pour les enregistrements allant de `0/1C40ED0` à la dernière position :

```
tests1=# \c tests1 - - 5432
tests1=# CREATE EXTENSION pg_walinspect;
CREATE EXTENSION
tests1=# SELECT start_lsn, xid, resource_manager, record_type, block_ref
  FROM pg_get_wal_records_info('0/1C44248', pg_current_wal_lsn()) \gx

-[ RECORD 1 ]----+-----------------------------------------------
start_lsn        | 0/1C44248
xid              | 748
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 2 ]----+-----------------------------------------------
start_lsn        | 0/1C44288
xid              | 748
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 3 ]----+-----------------------------------------------
start_lsn        | 0/1C442C8
xid              | 748
resource_manager | Transaction
record_type      | COMMIT
block_ref        |
-[ RECORD 4 ]----+-----------------------------------------------
start_lsn        | 0/1C442F8
xid              | 0
resource_manager | Standby
record_type      | RUNNING_XACTS
block_ref        |
-[ RECORD 5 ]----+-----------------------------------------------
start_lsn        | 0/1C44330
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 6 ]----+-----------------------------------------------
start_lsn        | 0/1C44370
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 7 ]----+-----------------------------------------------
start_lsn        | 0/1C443B0
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 8 ]----+-----------------------------------------------
start_lsn        | 0/1C443F0
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
```

```
-[ RECORD 9 ]----+-------------------------------------------------
start_lsn        | 0/1C44430
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 10 ]---+-------------------------------------------------
start_lsn        | 0/1C44470
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 11 ]---+-------------------------------------------------
start_lsn        | 0/1C444B0
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
-[ RECORD 12 ]---+-------------------------------------------------
start_lsn        | 0/1C444F0
xid              | 749
resource_manager | Btree
record_type      | INSERT_LEAF
block_ref        | blkref #0: rel 1663/16384/16388 fork main blk 1
-[ RECORD 13 ]---+-------------------------------------------------
start_lsn        | 0/1C44530
xid              | 749
resource_manager | Heap
record_type      | INSERT
block_ref        | blkref #0: rel 1663/16384/16385 fork main blk 0
[...]
```

Les trois premiers enregistrements concernent la transaction 748 (colonne `xid`). C'est bien cette transaction qui a ajouté la ligne 11, comme nous l'indiquait le résultat de la requête `DELETE` ainsi que le message dans les traces de PostgreSQL indiqué plus haut.

Le premier enregistrement indique une insertion (`record_type` à `INSERT` sur la table référencée 1663/16384/16389 (colonne `block_ref`). Le premier numéro est l'OID du tablespace, le deuxième numéro est l'OID de la base de données et le dernier numéro est le `refilenode` de la table. Il se trouve que la table `t1` a comme `relfilenode` 16389 :

```
tests1=# \c tests1 - - 5432
tests1=# SELECT relfilenode FROM pg_class WHERE relname='t1';
 relfilenode
-------------
       16389
(1 row)
```

Le deuxième enregistrement indique une écriture dans un index B-tree. Il s'agit de l'index lié à la clé primaire sur la table `t1`.

Enfin, le troisième enregistrement concerne la validation de la transaction. La transaction 748 s'arrête à l'emplacement `0/1C442F8`. Nous devons donc avancer le slot de réplication `sub1` à cet emplacement :

```
tests1=# \c tests1 - - 5432
tests1=# SELECT pg_replication_slot_advance('sub1', '0/1C442F8');
 pg_replication_slot_advance
-----------------------------
 (sub1,0/1C442F8)
(1 row)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
  8 |
  9 |
(9 rows)
```

Nous pouvons voir que la réplication a repris immédiatement et que les deux tables contiennent les mêmes données. Et on peut de nouveau ajouter des données :

```
tests1=# \c tests1 - - 5432
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5432".
tests1=# INSERT INTO t1 VALUES (-1), (-2);
INSERT 0 2
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
  8 |
  9 |
 -1 |
 -2 |
(11 rows)

tests1=# \c tests1 - - 5433
You are now connected to database "tests1" as user "postgres" via socket in "/tmp" at
 ↪  port "5433".
tests1=# TABLE t1;
 c1 | c2
----+----
  1 |  1
  2 |  2
```

```
  3 |  3
  4 |  4
  5 |
  6 |
  7 |
  8 |
  9 |
 -1 |
 -2 |
(11 rows)
```

8.5.12 Gérer les opérations de maintenance

- À faire séparément sur tous les serveurs
- `VACUUM`, `ANALYZE`, `REINDEX`

Dans la réplication physique, les opérations de maintenance ne sont réalisables que sur le serveur primaire, qui va envoyer le résultat de ces opérations aux serveurs secondaires.

Ce n'est pas le cas dans la réplication logique. Il faut bien voir les serveurs d'une réplication logique comme étant des serveurs indépendants les uns des autres.

Donc il faut configurer leur maintenance, avec les opérations `VACUUM`, `ANALYZE`, `REINDEX`, comme pour n'importe quel serveur PostgreSQL.

8.5.13 Gérer les sauvegardes & restaurations logiques

- `pg_dumpall` et `pg_dump`
  - sauvegardent publications et souscriptions
  - options `--no-publications` et `--no-subscriptions`
- Restauration d'une publication :
  - nouveau slot de réplication !
  - réconciliation de données à prévoir
- Restauration d'un abonnement :
  - `ENABLE` et `REFRESH PUBLICATION`
  - reprendre à zéro la copie… ou copier manuellement ?

Les sauvegardes logiques incluent les publications et souscriptions. Deux options (`--no-publications` et `--no-subscriptions`) permettent de les exclure.

Par contre, les slots de réplication liées aux publications, et leur position dans le flux de transaction, ne sont pas sauvegardés. Cela peut poser problème pour une restauration sans perte.

Après une restauration, il faudra soigneusement vérifier dans les traces que les réplications logiques ont repris leur fonctionnement, et qu'il n'y a pas de perte dans les données transmises.

Restauration d'une publication

Voici l'ordre SQL exécuté pour la restauration d'une publication complète :

```
CREATE PUBLICATION publi_complete FOR ALL TABLES
  WITH (publish = 'insert, update, delete');
```

Et ceux correspondant à la restauration d'une publication partielle :

```
CREATE PUBLICATION publi_partielle
  WITH (publish = 'insert, update, delete');
ALTER PUBLICATION publi_partielle ADD TABLE ONLY t1;
```

La publication sera fonctionnelle, mais il peut être délicat d'y raccrocher les abonnements existants. Selon ce qui s'est passé, le slot de réplication a souvent disparu, et il peut être plus simple de recréer ces abonnements.

Si le slot manque, le recréer sur l'instance d'origine est possible :

```
SELECT pg_create_logical_replication_slot ('abonnement','pgoutput') ;
```

Ces opérations sont obligatoirement manuelles. De toute façon, il faudra se poser la question de la resynchronisation des données. Généralement, l'origine aura été restaurée dans un état antérieur à celui déjà répliqué : les données répliquées à présent absentes de l'origine sont-elles toutes à conserver ? Comment gérer les clés primaires qui vont souvent entrer en conflit ?

Restauration d'une souscription

Pour une souscription, l'ordre SQL dans la sauvegarde est :

```
CREATE SUBSCRIPTION subscr_t3_2
  CONNECTION 'port=5444 user=logrepli dbname=b1'
  PUBLICATION publi_t3_2
  WITH (connect = false, slot_name = 'subscr_t3_2');
```

Contrairement à l'ordre exécuté manuellement à la création, celui-ci précise le nom du slot de réplication (au cas où il aurait été personnalisé) et désactive la connexion immédiate. Cette désactivation a pour effet de désactiver la souscription, de ne pas créer le slot de réplication et d'empêcher la copie initiale des données (dont nous n'avons pas besoin étant donné que nous les avons dans la sauvegarde, au moins en partie).

Une réplication restaurée est donc par défaut inactive.

Une fois la sauvegarde restaurée et les vérifications nécessaires effectuées, il est possible d'activer la souscription et de la rafraîchir :

```
ALTER SUBSCRIPTION subscr_complete ENABLE ;
ALTER SUBSCRIPTION subscr_complete REFRESH PUBLICATION ;
```

Ces opérations sont obligatoirement manuelles.

La restauration logique d'un abonnement revient à en créer un nouveau, et ne permet pas de savoir où la copie s'était arrêtée auparavant dans le flux des transactions : la copie des données sera intégralement relancée.
Sans autre opération, et si le contenu des tables répliquées a été restauré, le contenu déjà présent bloquera la réplication (s'il y a une clé primaire) ou de se retrouver en double (sans clé primaire). Il peut être plus simple de ne pas restaurer les données sur la destination, ou de tronquer les tables avant le `ENABLE`, pour reprendre la copie à zéro.
Une alternative est de ne pas effectuer la copie initiale :

```
ALTER SUBSCRIPTION nom_abonnement REFRESH PUBLICATION
WITH (copy_data = false) ;
```

auquel cas on risque d'avoir un « trou » entre les données restaurées et celles qui vont apparaître sur le publieur ; qu'il faudra corriger à la main dans les nombreux cas où cela est important.

8.5.14 Gérer les bascules & les restaurations physiques

Comme pour la réplication physique :

- Sauvegarde PITR
  - publications et souscriptions
  - slots ?
- Slots perdus et « trous » dans la réplication si :
  - bascule origine
  - restauration origine
  - restauration destination
- Contrôle délicat !
  - interdire les écritures à ces moments ?
- Bascule de la destination
  - si propre, devrait mieux se passer

Pendant ces opérations, il est fortement conseillé d'interdire les écritures dans les tables répliquées pour avoir une vision claire de ce qui a été répliqué et ne l'a pas été. Les slots doivent souvent être reconstruits, et il faut éviter que les tables soient modifiées entre-temps.

Restauration de l'instance d'origine :

Cela dépend de la méthode de sauvegarde/restauration utilisée, mais la restauration du serveur origine ne conserve généralement pas les slots de réplication (qui sont périmés de toute façon).

Il faudra recréer les slots, peut-être recréer les souscriptions, et pendant ce temps des trous dans les données répliquées peuvent apparaître, qu'il faudra vérifier ou corriger manuellement.

Bascule de l'instance d'origine :

Ici, l'instance d'origine est arrêtée et un de ses secondaires est promu comme nouveau serveur principal. Les slots de réplication étant propres à une instance, il ne seront pas disponibles immédiatement sur la nouvelle origine. Il faudra aussi reparamétrer la connexion des abonnements.

Il y a donc à nouveau un risque sérieux de perdre au moins quelques données répliquées.

Restauration de l'instance de destination :

Un slot de réplication sur l'origine garantit seulement que les journaux seront toujours disponibles pendant une indisponibilité du souscripteur. Ils ne permettent pas de revenir sur des données déjà répliquées.

En redémarrant, les abonnements vont tenter de se raccrocher au slot de réplication de l'origine, ce qui fonctionnera, mais ils ne recevront que des données jamais répliquées. Là encore des « trous »

dans les données répliquées peuvent apparaître si l'instance destination n'a pas été restaurée dans un état suffisamment récent !

Bascule de l'instance destination :

C'est le cas le plus favorable. Si la bascule s'est faite proprement sans perte entre l'ancienne destination et la nouvelle, il ne devrait pas y avoir de perte de données répliquées. Cela devra tout de même faire partie des contrôles.

8.5.15 Réplication logique depuis un secondaire comme origine

- Depuis PostgreSQL 16
- `wal_level = logical` sur le secondaire/origine
- Création de la publication toujours sur le primaire
- Le secondaire porte le slot et décode
- Latence supplémentaire

La réplication logique ne peut se faire depuis un serveur secondaire (lui-même en réplication physique) que si l'origine de la réplication logique (secondaires et primaire) est au moins sous PostgreSQL 16, mais pas forcément le destinataire.

La situation devient plus complexe car on a deux modes de réplication (physique et logique), et il faut bien distinguer les trois instances primaire, secondaire/origine et destination.

Rappelons que les slots de réplication sont propres à une instance, qu'elle soit secondaire ou pas. Le slot de réplication logique et le `walsender` associé seront donc créés sur le serveur secondaire, qui procédera au décodage logique, stockera les journaux au besoin, etc. et enverra les informations à l'instance destinataire.

Comme le secondaire est en lecture, il faudra continuer à créer et détruire les publications sur le primaire.

Latence :

Évidemment, la réplication logique sera tributaire des délais (voire pause) dans le rejeu des journaux sur le secondaire, et la latence en souffrira. En cas de complète inactivité, cette fonction, exécutée sur le primaire, permet d'envoyer dans les journaux le nécessaire pour une synchronisation des réplications logiques :

```
SELECT pg_log_standby_snapshot() ;
```

Promotion :

Si le serveur secondaire origine est promu et devient un primaire, la réplication logique qui y est attachée fonctionne toujours.

8.5.16 Combien de réplications logiques ?

- 1 publication logique = 1 walsender+1 slot par abonné
- Chaque worker doit décoder les WAL
  - Attention au CPU et à la RAM !
- Risques de slots bloqués
- Contournements :
  - regrouper les réplications
  - réplication depuis un secondaire
  - `streaming = on`

Il est possible de monter à plusieurs dizaines, voire une centaine, le nombre de réplications logiques depuis un même serveur origine.

Chaque publication nécessite donc un `walsender` et un slot par abonné sur la source. Sur la cible apparaît un `logical replication apply worker` pour chaque abonnement. D'autres processus peuvent aussi apparaître pendant la synchronisation (`table synchronization worker`) ou en cas d'application en parallèle des transactions (`parallel apply worker`).

On évitera donc de multiplier les réplications inutiles (par exemple en répliquant un schéma entier plutôt que chaque table séparément).

Il faudra parfois monter `max_wal_senders` et `max_replication_slots` sur le publieur, mais il n'y a pas besoin de monter `max_connections`. Sur la cible, il faudra vérifier `max_replication_slots`, `max_logical_replication_workers`, voire `max_worker_processes`. `max_sync_workers_per_subscription` ou `max_parallel_apply_workers_per_subscription` (initialisation et parallélisation) peuvent consommer encore d'autres workers. Prévoir donc de la marge.

Et il faut être conscient que chaque worker doit décoder le flux de journaux communs, ne serait-ce que pour chercher ce qui l'intéresse. Il y a donc un coût en CPU et en RAM, voire en disque lors de transactions longues. Dans ce dernier cas, il faudra arbitrer entre l'impact sur la RAM et la création de fichiers temporaires sur disque avec le paramètre `logical_decoding_work_mem`. et penser à activer l'option `streaming = on`.

S'il n'y a pas d'abonnement actif sur les tables répliquées, la consommation de ressources sera faible. Par contre, la présence de nombreux serveurs abonnés augmente le risque que certains slots bloquent le recyclage des journaux (penser à `max_slot_wal_keep_size`).

Depuis PostgreSQL 16, l'utilisation d'un serveur secondaire dédié est une option intéressante pour ce cas d'usage.

Voir cette discussion sur plgsql-general[7].

[7] https://www.postgresql.org/message-id/flat/CAAkB0aDof-atNom4qO_RGefgPDib3ukEzX1B9Tva11nusWMriA%40mail.gmail.com

8.6 SUPERVISION

- Méta-données
- Statistiques
- Outils

8.6.1 Catalogues systèmes - méta-données

- `pg_publication`
  - définition des publications
  - `\dRp` sous psql
- `pg_publication_tables`
  - tables ciblées par chaque publication
- `pg_subscription`
  - définition des souscriptions
  - `\dRs` sous psql

Dans la base origine :

Le catalogue système `pg_publication` contient la liste des publications, avec leur méta-données :

```
TABLE pg_publication ;
```

```
     pubname     | pubowner | puballtables | pubinsert | pubupdate | pubdelete
-----------------+----------+--------------+-----------+-----------+-----------
 publi_complete  |       10 | t            | t         | t         | t
 publi_partielle |       10 | f            | t         | t         | t
 publi_t3_1      |       10 | f            | t         | t         | t
```

Le catalogue système `pg_publication_tables` contient une ligne par table par publication :

```
TABLE pg_publication_tables ;
```

```
     pubname     | schemaname | tablename
-----------------+------------+-----------
 publi_complete  | public     | t1
```

```
 publi_complete  | public     | t3_1
 publi_complete  | public     | t3_2
 publi_complete  | public     | t2
 publi_complete  | public     | t3_3
 publi_complete  | public     | t4
 publi_partielle | public     | t1
 publi_partielle | public     | t2
 publi_t3_1      | public     | t3_1
```

On peut en déduire deux versions abrégées :

- la liste des tables par publication :

```
SELECT pubname, array_agg(tablename ORDER BY tablename) AS tables_list
FROM pg_publication_tables
GROUP BY pubname  ORDER BY pubname ;
```

```
     pubname     |         tables_list
-----------------+-----------------------------
 publi_complete  | {t1,t2,t3_1,t3_2,t3_3,t4,t5}
 publi_partielle | {t1,t2}
 publi_t3_1      | {t3_1}
```

- la liste des publications par table :

```
SELECT tablename, array_agg(pubname ORDER BY pubname) AS publications_list
FROM pg_publication_tables
GROUP BY tablename
ORDER BY tablename ;
```

```
 tablename |        publicationss_list
-----------+---------------------------------
 t1        | {publi_complete,publi_partielle}
 t2        | {publi_complete,publi_partielle}
 t3_1      | {publi_complete,publi_t3_1}
 t3_2      | {publi_complete}
 t3_3      | {publi_complete}
 t4        | {publi_complete}
 t5        | {publi_complete}
```

Dans la base destinataire :

Enfin, il y a aussi un catalogue système contenant la liste des souscriptions :

```
\x
Expanded display is on.
SELECT * FROM pg_subscription;
-[ RECORD 1 ]---+---------------------------------
subdbid         | 16443
subname         | subscr_t3_2
subowner        | 10
subenabled      | t
subconninfo     | port=5444 user=logrepli dbname=b1
subslotname     | subscr_t3_2
subsynccommit   | off
subpublications | {publi_t3_2}
```

8.6.2 Vues statistiques

- `pg_stat_replication`
  - statut de réplication
- `pg_replication_slots`
  - slots de réplication : statut
- `pg_stat_replication_slots` (v14)
  - volumes écrits/envoyés en *streaming* via les slots de réplication logique
- `pg_stat_subscription`
  - état des souscriptions
- `pg_replication_origin_status`
  - statut des origines de réplication
- `pg_stat_database_conflicts` (si origine est un secondaire)

Statut de la réplication :

Comme pour la réplication physique, le retard de réplication est visible ou calculable en utilisant les informations de la vue `pg_stat_replication` sur le serveur origine :

```
SELECT * FROM pg_stat_replication ;
```

```
-[ RECORD 1 ]----+------------------------------
pid              | 18200
usesysid         | 16442
usename          | logrepli
application_name | subscr_t3_1
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-12-20 10:31:01.13489+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/182D3C8
write_lsn        | 0/182D3C8
flush_lsn        | 0/182D3C8
replay_lsn       | 0/182D3C8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
```

```
sync_state       | async
-[ RECORD 2 ]----+------------------------------
pid              | 26606
usesysid         | 16442
usename          | logrepli
application_name | subscr_partielle
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-12-20 10:02:28.196654+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/182D3C8
write_lsn        | 0/182D3C8
flush_lsn        | 0/182D3C8
replay_lsn       | 0/182D3C8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
-[ RECORD 3 ]----+------------------------------
pid              | 15127
usesysid         | 16442
usename          | logrepli
application_name | subscr_complete
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2017-12-20 11:44:04.267249+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/182D3C8
write_lsn        | 0/182D3C8
flush_lsn        | 0/182D3C8
replay_lsn       | 0/182D3C8
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

La vue `pg_replication_slots` est complémentaire de `pg_stat_replication` car elle contient des statuts :

```
SELECT * FROM pg_replication_slots ;

-[ RECORD 1 ]-------+--------------------------
slot_name           | abonnement_16001
plugin              | pgoutput
slot_type           | logical
datoid              | 16388
database            | editeur
temporary           | f
active              | t
active_pid          | 1711279
```

```
xmin                |
catalog_xmin        | 2388
restart_lsn         | 1/FF7E0670
confirmed_flush_lsn | 1/FF7E06A8
wal_status          | reserved
safe_wal_size       |
two_phase           | f
conflicting         | f
```

Le dernier champ (apparu en version 16) indique une invalidation à cause d'un conflit de réplication.

Depuis la version 14, une autre vue, `pg_stat_replication_slots` (description complète dans la documentation[8], permet de suivre les volumétries (octets, nombre de transactions) écrites sur disque (*spilled*) ou envoyées en *streaming* :

```
SELECT * FROM pg_stat_replication_slots \gx

-[ RECORD 1 ]+-----------
slot_name    | abonnement
spill_txns   | 3
spill_count  | 7
spill_bytes  | 412435584
stream_txns  | 0
stream_count | 0
stream_bytes | 0
total_txns   | 30467
total_bytes  | 161694536
stats_reset  |
```

Souscriptions :

L'état des souscriptions est disponible sur les serveurs destination à partir de la vue `pg_stat_subscription` :

```
SELECT * FROM pg_stat_subscription ;

-[ RECORD 1 ]---------+------------------------------
subid                 | 16573
subname               | subscr_t3_2
pid                   | 18893
relid                 |
received_lsn          | 0/168A748
last_msg_send_time    | 2017-12-20 10:36:13.315798+01
last_msg_receipt_time | 2017-12-20 10:36:13.315849+01
latest_end_lsn        | 0/168A748
latest_end_time       | 2017-12-20 10:36:13.315798+01
```

Conflits de réplication :

Depuis PostgreSQL 16, un secondaire peut être origine d'une réplication logique. Comme dans une réplication physique classique, il est possible d'avoir des conflits de réplication (le primaire envoie des modifications sur des lignes que le secondaire aurait voulu garder pour ses abonnés).

[8] https://docs.postgresql.fr/current/monitoring-stats.html#MONITORING-PG-STAT-REPLICATION-SLOTS-VIEW

Dans une réplication physique classique, le conflit entraîne juste l'arrêt de requêtes sur le secondaire. Mais si le secondaire est origine d'une réplication logique, celle-ci peut décrocher. Le problème apparaît surtout lors d'une modification dans le schéma de données. Le message suivant apparaît dans les traces de la destination si un slot de réplication a été invalidé suite à ce conflit :

```
LOG:  logical replication apply worker for subscription "abonnement_decompte" has
↪  started
ERROR:  could not start WAL streaming: ERROR:  can no longer get changes from
↪  replication slot "decompte_abonnement_16001"
DETAIL:  This slot has been invalidated because it was conflicting with recovery.
```

L'option `disable_on_error` sur la souscription permet d'éviter qu'elle ne tente de se reconnecter en boucle. Le plus propre est de sécuriser la réplication entre primaire et secondaire en passant `hot_standby_feedback` à `on` sur le secondaire (ce qui doit toujours se sécuriser sur le primaire en mettant un seuil dans `max_slot_wal_keep_size`).

8.6.3 Outils de supervision

- `check_pgactivity`
  - `replication_slots`
- `check_postgres`
  - `same_schema`

Il est possible de surveiller le retard de réplication via l'état des slots de réplication, comme le propose l'outil `check_pgactivity` (disponible sur github[9] ou les paquets des dépôts). Ici, il n'y a pas de retard sur la réplication, pour les trois slots :

```
$ ./check_pgactivity -s replication_slots -p 5441 -F human

Service        : POSTGRES_REPLICATION_SLOTS
Returns        : 0 (OK)
Message        : Replication slots OK
Perfdata       : subscr_complete_wal=0File
Perfdata       : subscr_complete_spilled=0File
Perfdata       : subscr_t3_1_wal=0File
Perfdata       : subscr_t3_1_spilled=0File
Perfdata       : subscr_partielle_wal=0File
Perfdata       : subscr_partielle_spilled=0File
```

Faisons quelques insertions après l'arrêt de s3 (qui correspond à la souscription pour la réplication partielle) :

[9] https://github.com/OPMDG/check_pgactivity

```
INSERT INTO t1 SELECT generate_series(1000000, 2000000);
```

L'outil détecte bien que le slot `subscr_partielle` a un retard conséquent (8 journaux de transactions) et affiche le nombre de fichiers de débordement créés :

```
$ ./check_pgactivity -s replication_slots -p 5441 -F human

Service        : POSTGRES_REPLICATION_SLOTS
Returns        : 0 (OK)
Message        : Replication slots OK
Perfdata       : subscr_t3_1_wal=8File
Perfdata       : subscr_t3_1_spilled=0File
Perfdata       : subscr_partielle_wal=8File
Perfdata       : subscr_partielle_spilled=9File
Perfdata       : subscr_complete_wal=8File
Perfdata       : subscr_complete_spilled=9File
```

Il est aussi possible d'utiliser l'action `same_schema` avec l'outil `check_postgres` (disponible aussi sur github[10]) pour détecter des différences de schémas entre deux serveurs (l'origine et une destination).

[10] https://github.com/bucardo/check_postgres/

8.7 MIGRATION MAJEURE PAR RÉPLICATION LOGIQUE

- Possible entre versions 10 et supérieures
- Remplace Slony, Bucardo…
- Bascule très rapide
- Et retour possible
- Des limitations

La réplication logique rend possible une migration entre deux instances de version majeure différente avec une indisponibilité très courte. La base à migrer doit bien sûr être en version 10 ou supérieure. C'était déjà possible avec des outils de réplication par trigger comme Slony ou Bucardo. Ces outils externes ne sont maintenant plus nécessaires. (Noter que Slony en particulier reste parfaitement utilisable et recommandable, et sert encore pour nombre de migrations).

Le principe est de répliquer une base à l'identique alors que la production tourne. Lors de la bascule, il suffit d'attendre que les dernières données soient répliquées, ce qui peut être très rapide, et de connecter les applications au nouveau serveur. La réplication peut alors être inversée pour garder l'ancienne production synchrone, permettant de rebasculer dessus en cas de problème sans perdre les données modifiées depuis la bascule.

Les étapes sont :

- copie des structures et des objets globaux concernés ;
- mise en place d'une publication sur la source et d'un abonnement sur la cible ;
- suivi de la réplication (*lag* entre les serveurs) ;
- arrêt des connexions applicatives ;
- attente de la fin de la réplication logique ;
- isolation de la base source des connexions applicatives ;
- synchronisation manuelle des valeurs des séquences (non répliquées) ;
- suppression de la publication et de l'abonnement ;
- éventuellement création d'un abonnement et d'une publication en sens inverse ;
- ouverture de la base cible aux applications.

Les restrictions liées à la réplication logique subsistent :

- les modifications de schéma effectuées pendant la synchronisation ne sont pas répliquées (cela est problématique si l'application elle-même effectue du DDL sur des tables non temporaires) ;
- les `TRUNCATE` depuis une base v10 ne sont pas répliqués ;
- les *Large objects* et les séquences ne sont pas répliqués ;
- il est fortement conseillé que toutes les tables aient des clés primaires ;
- la réplication fonctionnant uniquement pour les tables « de base », les vues matérialisées sont à reconstruire sur la cible ;
- jusqu'en version 13, le partitionnement doit être identique des deux côtés.

Cette méthode reste donc plus complexe et fastidieuse qu'une migration par `pg_dump` / `pg_restore` ou `pg_upgrade`.

8.8 RAPPEL DES LIMITATIONS DE LA RÉPLICATION LOGIQUE NATIVE

- Pas de réplication : DDL, LO, valeurs de séquence
- Pas de réplication des tables partitionnées (< v13)
  - mais réplication possible des partitions
- Pas de réplication vers une table partitionnée (< v13)
- Contraintes d'unicité obligatoires pour les `UPDATE` / `DELETE`
- Coût CPU, disque, RAM
- Réplication déclenchée uniquement lors du `COMMIT` (< v14)
- Que faire lors des restaurations/bascules ?

Rappelons que la réplication logique native ne réplique pas les ordres DDL. Elle se base uniquement au niveau des données (donc les ordres DML, et `TRUNCATE`). Les valeurs des séquences et les Larges Objects ne sont pas répliqués.

Avant la version 13, il n'était pas possible d'ajouter une table partitionnée à une publication pour qu'elle et ses partitions soient répliquées. Il fallait ajouter chaque partition individuellement. Cette limitation a été supprimée en version 13. Toujours avant la version 13, il n'était pas possible d'envoyer des données vers une table partitionnée.

Pour les versions inférieures à 14, la réplication logique n'est déclenchée que lors d'un `COMMIT`, avec un délai de réplication pour les transactions longues. Pensez à `streaming=on`.

Enfin, la réplication logique doit tenir compte des cas de restauration, ou bascule, d'une des instances impliquées. Le concept de flux unique de transaction unique ne s'applique plus ici, et il n'est pas prévu de moyen pour garantir que la réplication se fera sans aucune perte ou risque de doublon. La mise en place de la réplication logique doit toujours prévoir ce qu'il faudra faire dans ce cas.

Certaines applications supporteront cette limite. Dans d'autres, il sera plus ou moins facile de reprendre la réplication à zéro. Parfois, une réconciliation manuelle sera nécessaire (la présence de clés primaires peut grandement aider). Dans certains cas, ce problème peut devenir bloquant ou réclamer des développements.

8.9 OUTILS DE RÉPLICATION LOGIQUE EXTERNE

- Conseillé : Slony
- Non conseillés: Londiste, Bucardo

Slony est un outil que nous utilisons régulièrement pour des montées de versions majeures.

Nous n'avons pas rencontré Londiste et Bucardo en production depuis plusieurs années. Ils semblent encore maintenus mais le développement s'est depuis longtemps figé, et nous ne conseillons pas leur utilisation en production.

8.9.1 Slony : Carte d'identité

- Projet libre (`BSD`)
- Asynchrone / Asymétrique
- Diffusion des résultats (triggers)

Slony[11] est un très ancien projet libre de réplication pour PostgreSQL. C'était l'outil de choix avant l'arrivée de la réplication native dans PostgreSQL.

8.9.2 Slony : Fonctionnalités

- Réplication de tables sélectionnées
- Procédures de bascule
  - *switchover* / *switchback*
  - *failover* / *failback*

Slony permet de choisir les tables à répliquer. Il faudra ajouter à la réplication toute nouvelle table qui serait créée après sa mise en place.

[11] http://slony.info/

Les procédures de bascule chez Slony sont très simples. Il est ainsi possible de basculer un serveur primaire et son serveur secondaire autant de fois qu'on le souhaite, très rapidement, sans avoir à reconstruire quoi que ce soit.

8.9.3 Slony : Technique

- Réplication basée sur des triggers
- Démons externes, écrits en C
- Le primaire est un **provider**
- Les secondaires sont des **subscribers**

Slony est un système de réplication asynchrone/asymétrique, donc un seul primaire et un ou plusieurs serveurs secondaires mis à jour à intervalle régulier. La récupération des données modifiées se fait par des triggers, qui stockent les modifications dans des tables propres à Slony avant leur transfert vers les secondaires. Un système de démon récupère les données pour les envoyer sur les secondaires et les applique.

Les démons et les triggers sont écrits en C, ce qui permet à Slony d'être très performant.

Au niveau du vocabulaire utilisé, le primaire est souvent appelé un « provider » (il fournit les données aux serveurs secondaires) et les secondaires sont souvent des « subscribers » (ils s'abonnent au flux de réplication pour récupérer les données modifiées).

8.9.4 Slony : Points forts

- Choix des tables et séquences à répliquer
- Indépendance des versions de PostgreSQL
- Technique de propagation des DDL
- Robustesse

Slony dispose de nombreux points forts, parfois liés au simple fait qu'il s'agit d'une réplication logique.

Il permet de ne répliquer qu'un sous-ensemble des objets d'une instance : pas forcément toutes les bases, pas forcément toutes les tables d'une base particulière, etc.

Le serveur primaire et les serveurs secondaires n'ont pas besoin d'utiliser la même version majeure de PostgreSQL. Il est donc possible de mettre à jour en plusieurs étapes (plutôt que tous les serveurs à la fois). Cela facilite aussi le passage à une version majeure ultérieure.

Même si la réplication des DDL est impossible, leur envoi aux différents serveurs est possible grâce à un outil fourni. Tous les systèmes de réplication par triggers ne peuvent pas en dire autant.

8.9.5 Slony : Limites

- Le réseau doit être fiable : peu de *lag*, pas ou peu de coupures
- Supervision délicate
- Modifications de schémas complexes

Slony peut survivre avec un réseau coupé. Cependant, il n'aime pas quand le réseau passe son temps à être disponible puis indisponible. Les démons slon ont tendance à croire qu'ils sont toujours connectés alors que ce n'est plus le cas.

Superviser Slony n'est possible que via une table statistique appelée `sl_status`. Elle fournit principalement deux informations : le retard en nombre d'événements de synchronisation et la date de la dernière synchronisation.

Enfin, la modification de la structure d'une base, même si elle est simplifiée avec le script fourni, n'est pas simple, en tout cas beaucoup moins simple que d'exécuter une requête DDL seule.

8.9.6 Slony : Utilisations

- Réplications complexes
- Infocentre (*many to one*)
- Bases spécialisées (recherche plein texte, traitements lourds, etc.)
- Migrations de versions majeures avec indisponibilité réduite

Bien que la réplication logique soit arrivée avec PostgreSQL 10, Slony garde son utilité pour les nombreuses instances des versions précédentes.

Slony peut se révéler intéressant car il est possible d'avoir des tables de travail en écriture sur le secondaire avec Slony. Il est aussi possible d'ajouter des index sur le secondaire qui ne seront pas présents sur le serveur primaire (on évite donc la charge de maintenance des index par le serveur primaire, tout en permettant de bonnes performances pour la création des rapports).

Il est encore fréquent d'utiliser Slony pour des migrations entre deux versions majeures avec une indisponibilité réduite, voire avec un retour en arrière possible.

Pour plus d'informations sur Slony, n'hésitez pas à lire un de nos articles disponibles sur notre site[12]. Le thème des réplications complexes a aussi été abordé lors du PostgreSQL Sessions 2012[13].

[12] https://www.dalibo.org/hs44_slony_la_replication_des_donnees_par_trigger
[13] https://www.postgresql-sessions.org/assets/archives/pgsessions3_slony.pdf

8.10 CONCLUSION

- Réplication logique simple et pratique
  - …avec ses subtilités

La réplication logique de PostgreSQL apparue en version 10 continue de s'améliorer avec les versions. Elle complète la réplication physique sans la remplacer.

Les cas d'utilisation sont nombreux, mais la supervision est délicate et il faut prévoir les sauvegardes/restaurations et bascules.

8.10.1 Questions

N'hésitez pas, c'est le moment !

8.11 QUIZ

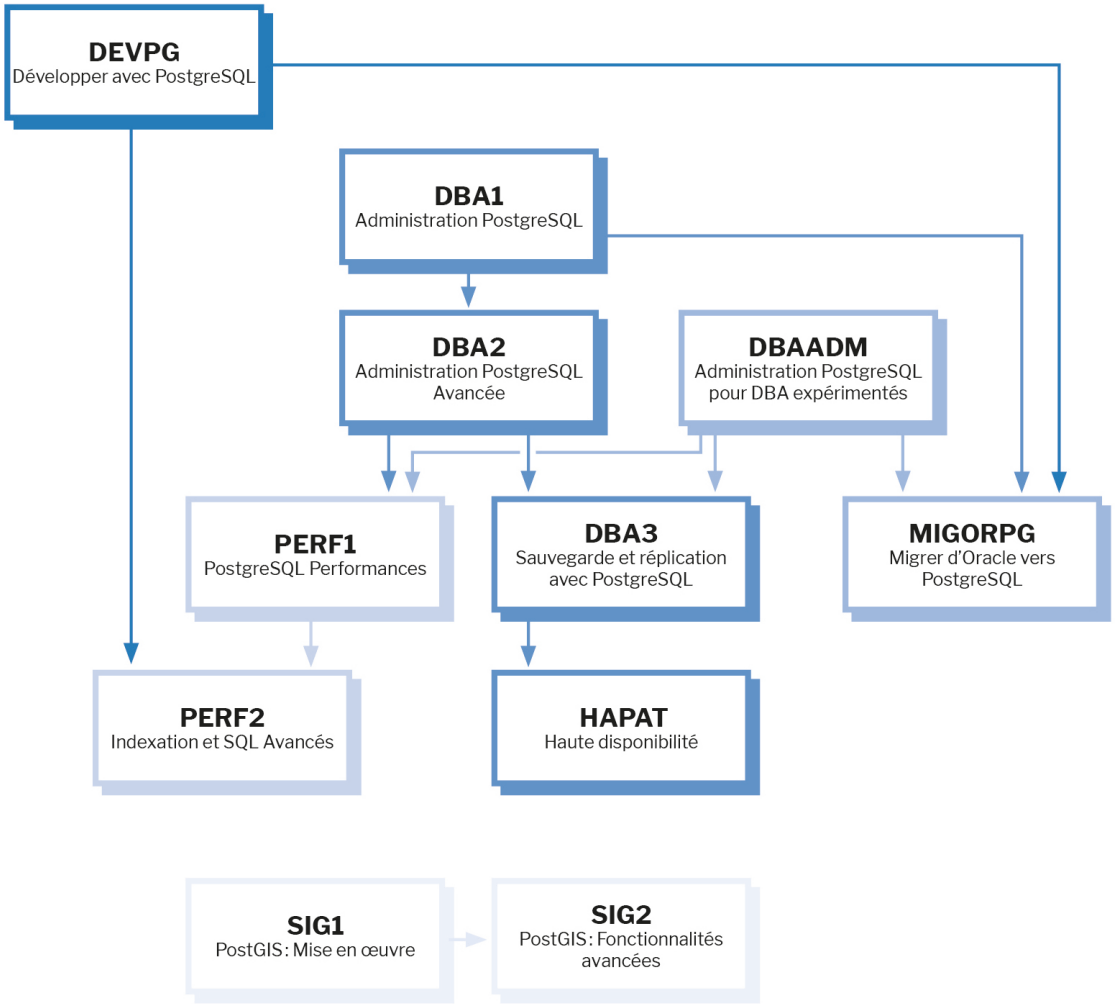
https://dali.bo/w5_quiz

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.