Module I2

Point In Time Recovery

24.04

Table des matières

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

1/ Sauvegarde physique à chaud et PITR **5**
1.1 Introduction . . . 6
1.1.1 Au menu . . . 6
1.2 PITR . . . 8
1.2.1 Principes . . . 8
1.2.2 Avantages . . . 9
1.2.3 Inconvénients . . . 10
1.3 Copie physique à chaud ponctuelle avec pg_basebackup . . . 11
1.4 Sauvegarde PITR . . . 14
1.4.1 Méthodes d'archivage . . . 14
1.4.2 Choix du répertoire d'archivage . . . 15
1.4.3 Processus archiver : configuration . . . 15
1.4.4 Processus archiver : lancement . . . 19
1.4.5 Processus archiver : supervision . . . 19
1.4.6 pg_receivewal . . . 21
1.4.7 pg_receivewal - configuration serveur . . . 23
1.4.8 pg_receivewal - redémarrage du serveur . . . 23
1.4.9 pg_receivewal - lancement de l'outil . . . 24
1.4.10 Avantages et inconvénients . . . 25
1.5 Sauvegarde PITR manuelle . . . 26
1.5.1 Sauvegarde manuelle - 1/3 : pg_backup_start . . . 27
1.5.2 Sauvegarde manuelle - 2/3 : copie des fichiers . . . 28
1.5.3 Sauvegarde manuelle - 3/3 : pg_backup_stop . . . 30
1.5.4 Sauvegarde de base à chaud : pg_basebackup . . . 31
1.5.5 Fréquence de la sauvegarde de base . . . 32
1.5.6 Suivi de la sauvegarde de base . . . 32
1.6 Restaurer une sauvegarde PITR . . . 33
1.6.1 Restaurer une sauvegarde PITR (1/5) . . . 33
1.6.2 Restaurer une sauvegarde PITR (2/5) . . . 33
1.6.3 Restaurer une sauvegarde PITR (3/5) . . . 34
1.6.4 Restaurer une sauvegarde PITR (4/5) . . . 35
1.6.5 Restaurer une sauvegarde PITR (5/5) . . . 37
1.6.6 Restauration PITR : durée . . . 38

1.6.7 Restauration PITR : différentes timelines 39
1.6.8 Restauration PITR : illustration des timelines 41
1.6.9 Après la restauration 43
1.7 Pour aller plus loin 44
1.7.1 Réduire le nombre de journaux sauvegardés 44
1.7.2 Compresser les journaux de transactions 45
1.7.3 Outils de sauvegarde PITR dédiés 46
1.7.4 pgBackRest 47
1.7.5 barman 48
1.7.6 pitrery 49
1.8 Conclusion 50
1.8.1 Questions 50
1.9 Quiz 51
1.10 Travaux pratiques 52
1.10.1 pg_basebackup : sauvegarde ponctuelle & restauration 52
1.10.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants 53
1.11 Travaux pratiques (solutions) 55
1.11.1 pg_basebackup : sauvegarde ponctuelle & restauration 55
1.11.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants 59

Les formations Dalibo **65**
Cursus des formations 65
Les livres blancs 66
Téléchargement gratuit 66

Sur ce document

Formation	Module I2
Titre	Point In Time Recovery
Révision	24.04
PDF	https://dali.bo/i2_pdf
EPUB	https://dali.bo/i2_epub
HTML	https://dali.bo/i2_html
Slides	https://dali.bo/i2_slides
TP	https://dali.bo/i2_tp
TP (solutions)	https://dali.bo/i2_solutions

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

Chers lectrices & lecteurs,

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

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

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

À propos de DALIBO

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

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

[1] mailto:formation@dalibo.com

Remerciements

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

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

Forme de ce manuel

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

Licence Creative Commons CC-BY-NC-SA

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

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

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

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

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

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

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

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

Marques déposées

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

Versions de PostgreSQL couvertes

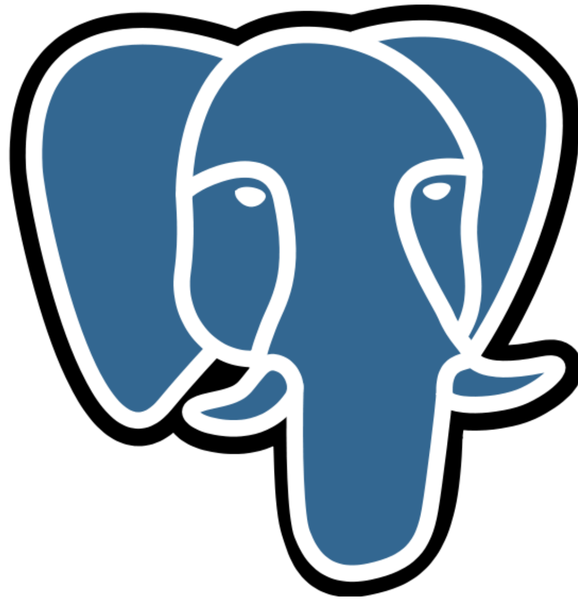
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/ Sauvegarde physique à chaud et PITR



1.1 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*).

1.1.1 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.

1.2 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.

1.2.1 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).

1.2.2 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.

1.2.3 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.

1.3 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

1.4 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.

1.4.1 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.

1.4.2 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
```

1.4.3 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.

1.4.4 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.

1.4.5 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  | 0000000200000001000000D9
last_archived_time | 2020-01-17 18:26:03.715946+00
failed_count       | 6
last_failed_wal    | 0000000200000001000000D7
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/0000000200000001000000D7 /opt/pgsql/archives/0000000200000001000000D7
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/0000000200000001000000D7 /opt/pgsql/archives/0000000200000001000000D7
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
--------------------------------
 0000000200000001000000DE.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
```

1.4.6 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/>
```

1.4.7 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.

1.4.8 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`.

1.4.9 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 000000010000000000000010.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.

1.4.10 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.

1.5 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.

1.5.1 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…).

1.5.2 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

1.5.3 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 00000001000000220000002B)+
            | 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 00000001000000220000002B)
STOP WAL LOCATION: 22/2FE5C788 (file 00000001000000220000002F)
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.

1.5.4 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.

1.5.5 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.

1.5.6 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.

1.6 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.

1.6.1 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.

1.6.2 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.

1.6.3 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 parallélisation 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.

1.6.4 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].

1.6.5 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 "00000001000000010000003C"
                                         from archive
2020-01-17 16:08:46.116 UTC [15221] LOG: restored log file
 ↪ "00000001000000010000003D"…
2020-01-17 16:08:46.547 UTC [15221] LOG: restored log file
 ↪ "00000001000000010000003E"…
2020-01-17 16:08:47.262 UTC [15221] LOG: restored log file
 ↪ "00000001000000010000003F"…
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
 ↪ "00000001000000010000003F"…
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`.

1.6.6 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.

1.6.7 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/
00000001000000010000003C
00000001000000010000003D
00000001000000010000003E
00000001000000010000003F
000000010000000100000040
00000002.history
```

```
00000002000000010000003F
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.

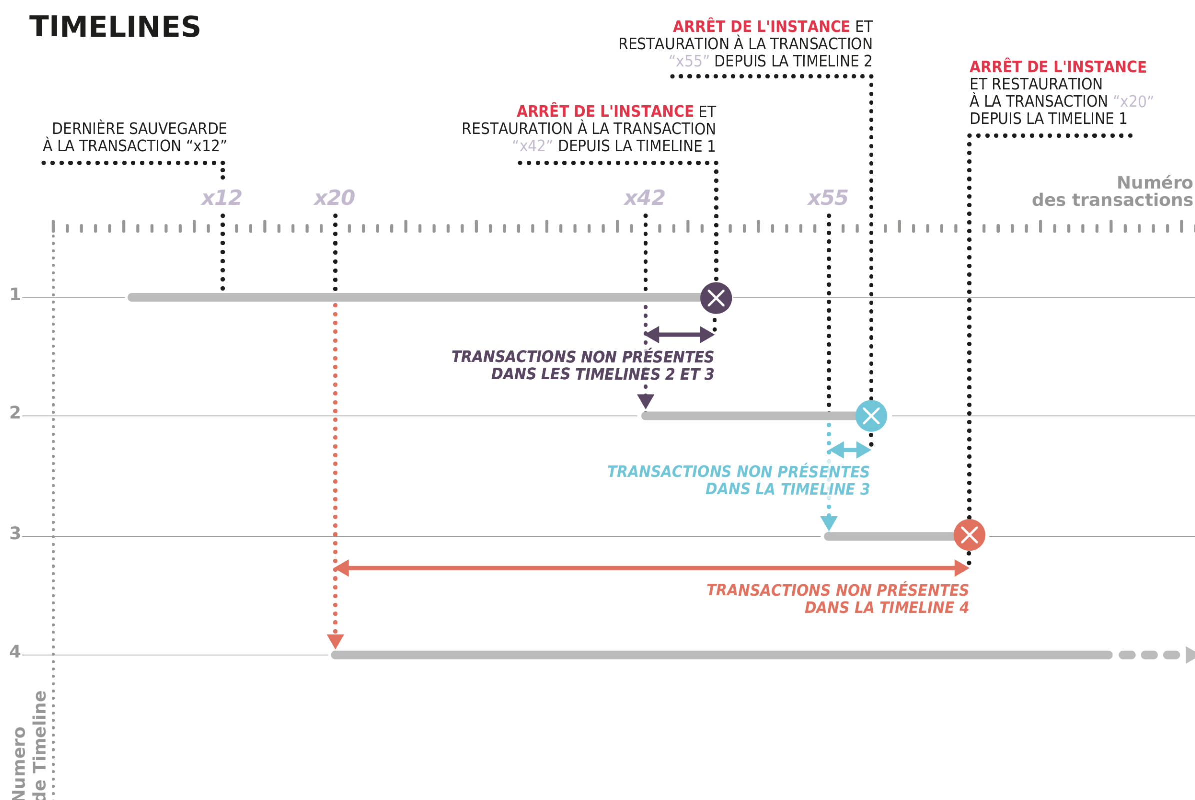
Pour choisir une autre *timeline* que la dernière, il faut donner le numéro de la *timeline* cible comme valeur du paramètre `recovery_target_timeline`. Les timelines permettent d'effectuer plusieurs restaurations successives à partir du même *base backup* et d'archiver au même endroit sans mélanger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

1.6.8 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.

1.6.9 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.

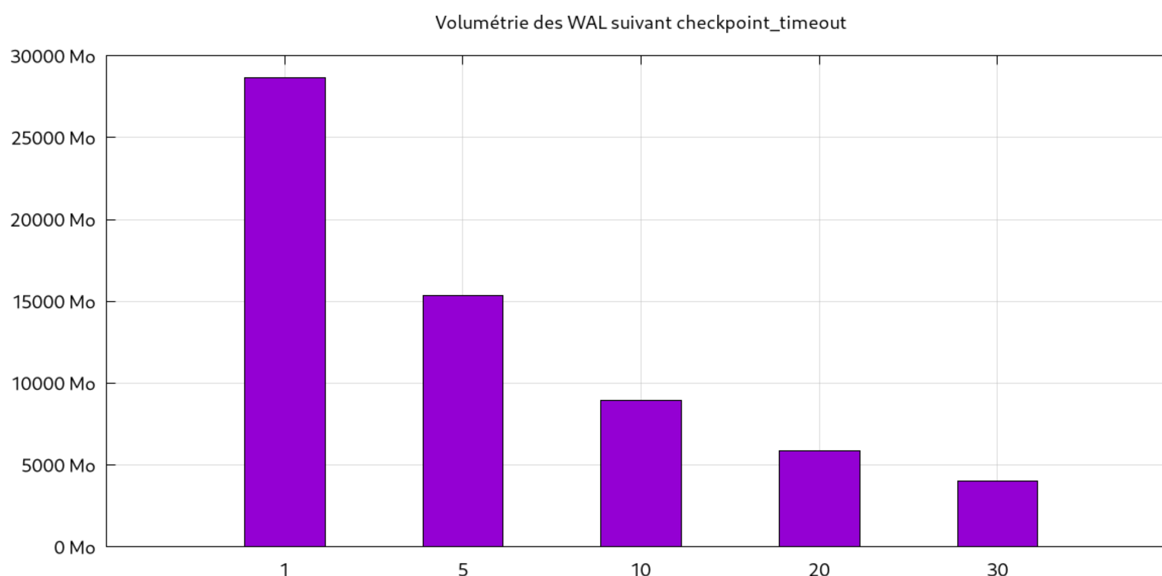
La copie de journaux à la main est donc risquée. Il ne faut pas hésiter à reprendre une sauvegarde complète du nouveau primaire pour repartir d'une base sûre.

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.)

1.7 POUR ALLER PLUS LOIN

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

1.7.1 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 giga-octets.

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.

1.7.2 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.

1.7.3 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.

1.7.4 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].

1.7.5 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/

1.7.6 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/

1.8 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).

1.8.1 Questions

N'hésitez pas, c'est le moment !

1.9 QUIZ

https://dali.bo/i2_quiz

1.10 TRAVAUX PRATIQUES

1.10.1 pg_basebackup : sauvegarde ponctuelle & restauration

But : Créer une sauvegarde physique à chaud à un moment précis de la base avec `pg_basebackup`, et la restaurer.

Configurer la réplication dans `postgresql.conf` et `pg_hba.conf` :

- désactiver l'archivage s'il est actif
- autoriser des connexions de réplication en streaming en local.

Pour insérer des données :

- générer de l'activité avec `pgbench` en tant qu'utilisateur **postgres** :

```
$ createdb bench
$ /usr/pgsql-16/bin/pgbench -i -s 100 bench
$ /usr/pgsql-16/bin/pgbench bench -n -P 5 -R 20 -T 720
```

- laisser tourner en arrière-plan
- surveiller l'évolution de l'activité sur la table `pgbench_history`, par exemple ainsi :

```
$ watch -n 1 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history ;'"
```

En parallèle, sauvegarder l'instance avec :

- `pg_basebackup` au format tar, compressé avec gzip ;
- sans oublier les journaux ;
- avec l'option `--max-rate=16M` pour ralentir la sauvegarde ;
- le répertoire de sauvegarde sera `/var/lib/pgsql/16/backups/basebackup` ;
- surveillez la progression dans une autre session avec la vue système adéquate.

Une fois la sauvegarde terminée :

- regarder les fichiers générés ;
- arrêter la session `pgbench` ; Afficher la date de dernière modification dans `pgbench_history`.

- Arrêter l'instance.

- Faire une copie à froid des données (par exemple avec `cp -rfp`) vers `/var/lib/pgsql/16/data.old` (cette copie resservira plus tard).

- Vider le répertoire des données.
- Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.
- Redémarrer l'instance.

Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

Quelle est la dernière donnée restaurée ?

Tenter une nouvelle restauration depuis l'archive `pg_basebackup` *sans* restaurer les journaux de transaction. Que se passe-t-il ?

1.10.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants

But : Coupler une sauvegarde à chaud avec `pg_basebackup` et l'archivage

Remettre en place la copie à froid de l'instance prise précédemment dans `/var/lib/pgsql/16/data.old`. Configurer l'archivage vers un répertoire `/var/lib/pgsql/16/archives`, par exemple avec `rsync`. Configurer la commande de restauration inverse. Démarrer PostgreSQL.

Générer à nouveau de l'activité avec `pgbench`. Vérifier que l'archivage fonctionne.

En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup` au format plain.

Utiliser `pg_verify_backup` pour contrôler l'intégrité de la sauvegarde.

À quoi correspond le fichier finissant par `.backup` dans les archives ?

Arrêter pgbench et noter la date des dernières données insérées.

> Effacer le PGDATA. Restaurer la sauvegarde précédente *sans* les journaux. Configurer la `restore_command`. Créer le fichier `recovery.signal`. Démarrer PostgreSQL.

> Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

> Vérifier quelles données ont été restaurées.

1.11 TRAVAUX PRATIQUES (SOLUTIONS)

1.11.1 pg_basebackup : sauvegarde ponctuelle & restauration

Dans ce qui suit, la plupart des commandes seront à lancer en tant que **postgres**, les ordres `sudo` nécessitant un utilisateur privilégié.

> Configurer la réplication dans `postgresql.conf` et `pg_hba.conf` :
>
> - désactiver l'archivage s'il est actif
> - autoriser des connexions de réplication en streaming en local.

On n'aura ici pas besoin de l'archivage. S'il est déjà actif, on peut se contenter d'inhiber ainsi la commande d'archivage :

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

(Cela permet d'épargner le redémarrage à chaque modification de `archive_mode`.)

Vérifier la configurer de l'autorisation de connexion en réplication dans `pg_hba.conf`. Si besoin, mettre à jour la ligne en fin de fichier :

```
local   replication     all                                     peer
```

Cela va ouvrir l'accès sans mot de passe depuis l'utilisateur système **postgres**.

Redémarrer PostgreSQL :

```
sudo systemctl restart postgresql-16
```

> Pour insérer des données :
>
> - générer de l'activité avec `pgbench` en tant qu'utilisateur **postgres** :
>
> ```
> $ createdb bench
> $ /usr/pgsql-16/bin/pgbench -i -s 100 bench
> $ /usr/pgsql-16/bin/pgbench bench -n -P 5 -R 20 -T 720
> ```
>
> - laisser tourner en arrière-plan
> - surveiller l'évolution de l'activité sur la table `pgbench_history`, par exemple ainsi :
>
> ```
> $ watch -n 1 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history ;'"
> ```

> En parallèle, sauvegarder l'instance avec :
>
> - `pg_basebackup` au format tar, compressé avec gzip ;
> - sans oublier les journaux ;
> - avec l'option `--max-rate=16M` pour ralentir la sauvegarde ;
> - le répertoire de sauvegarde sera `/var/lib/pgsql/16/backups/basebackup` ;

> - surveillez la progression dans une autre session avec la vue système adéquate.

En tant que **postgres** :

```
pg_basebackup -D /var/lib/pgsql/16/backups/basebackup -Ft \
--checkpoint=fast --gzip --progress --max-rate=16M
```

```
1583675/1583675 kB (100%), 1/1 tablespace
```

La progression peut se suivre depuis psql avec :

```
\x on
SELECT * FROM pg_stat_progress_basebackup ;
\watch
```

```
Thu Nov 11 16:58:05 2023 (every 2s)

-[ RECORD 1 ]--------+---------------------------------
pid                 | 19763
phase                | waiting for checkpoint to finish
backup_total         |
backup_streamed      | 0
tablespaces_total    | 0
tablespaces_streamed | 0

Thu Nov 11 16:58:07 2023 (every 2s)

-[ RECORD 1 ]--------+-------------------------
pid                  | 19763
phase                | streaming database files
backup_total         | 1611215360
backup_streamed      | 29354496
tablespaces_total    | 1
tablespaces_streamed | 0
…
```

Évidemment, en production, il ne faut pas sauvegarder en local.

> Une fois la sauvegarde terminée :
>
> - regarder les fichiers générés ;
> - arrêter la session `pgbench` ; Afficher la date de dernière modification dans `pgbench_history`.

```
$ ls -lha /var/lib/pgsql/16/backups/basebackup

…
-rw-------. 1 postgres postgres 180K Nov 11 17:00 backup_manifest
-rw-------. 1 postgres postgres  91M Nov 11 17:00 base.tar.gz
-rw-------. 1 postgres postgres  23M Nov 11 17:00 pg_wal.tar.gz
```

On obtient donc :

- une archive de la sauvegarde « de base » ;
- une archive des journaux nécessaires ;
- un fichier manifeste au format texte contenant les sommes de contrôles des fichiers archivés.

`pgbench` s'arrête avec un simple **Ctrl-C**. L'heure de dernière modification est :

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
----------------------------
 2023-11-05 17:01:51.595414
```

> - Arrêter l'instance.
> - Faire une copie à froid des données (par exemple avec `cp -rfp`) vers `/var/lib/pgsql/16/data.old` (cette copie resservira plus tard).

En tant qu'utilisateur privilégié :

```
sudo systemctl stop postgresql-16
```

En tant que **postgres** :

```
cp -rfp /var/lib/pgsql/16/data /var/lib/pgsql/16/data.old
```

> - Vider le répertoire des données.
> - Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.
> - Redémarrer l'instance.

On restaure dans le répertoire de données l'archive de base, puis les journaux dans leur sous-répertoire. La suppression des traces est optionnelle, mais elle nous permettra de ne pas mélanger celles d'avant et d'après la restauration.

En tant que **postgres** :

```
rm -rf /var/lib/pgsql/16/data/*
tar -C /var/lib/pgsql/16/data \
    -xzf /var/lib/pgsql/16/backups/basebackup/base.tar.gz
tar -C /var/lib/pgsql/16/data/pg_wal \
    -xzf /var/lib/pgsql/16/backups/basebackup/pg_wal.tar.gz
rm -rf /var/lib/pgsql/16/data/log/*
```

```
sudo systemctl start postgresql-16
```

> Une fois l'instance restaurée et démarrée, vérifier les traces : la base doit accepter les connexions.

```
tail -F /var/lib/pgsql/16/data/log/postgresql-*.log
```

```
…
… LOG:  database system was interrupted; last known up at 2023-11-05 16:59:03 UTC
… LOG:  redo starts at 0/830000B0
… LOG:  consistent recovery state reached at 0/8E8450F0
… LOG:  redo done at 0/8E8450F0 system usage: CPU: user: 0.28 s, system: 0.24 s,
 ↪  elapsed: 0.59 s
… LOG:  checkpoint starting: end-of-recovery immediate wait
… LOG:  checkpoint complete: wrote 16008 buffers (97.7%); …
… LOG:  database system is ready to accept connections
```

PostgreSQL considère qu'il a été interrompu brutalement et part en *recovery*. Noter en particulier la mention *consistent recovery state reached* : la sauvegarde est bien cohérente.

> Quelle est la dernière donnée restaurée ?

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
----------------------------
 2023-11-05 17:00:40.936925
```

Grâce aux journaux (`pg_wal`) restaurés, l'ensemble des modifications survenues **pendant** la sauvegarde ont bien été récupérées. Par contre, les données générées après la sauvegarde n'ont, elles, pas été récupérées.

> Tenter une nouvelle restauration depuis l'archive `pg_basebackup` *sans* restaurer les journaux de transaction. Que se passe-t-il ?

```
sudo systemctl stop postgresql-16
```

```
rm -rf /var/lib/pgsql/16/data/*
tar -C /var/lib/pgsql/16/data \
    -xzf /var/lib/pgsql/16/backups/basebackup/base.tar.gz
rm -rf /var/lib/pgsql/16/data/log/*
systemctl start postgresql-16
```

```
sudo systemctl start postgresql-16
```

Résultat :

```
Job for postgresql-16.service failed because the control process exited with error
↪   code.
See "systemctl status postgresql-16.service" and "journalctl -xe" for details.
```

Pour trouver la cause :

```
tail -F /var/lib/pgsql/16/data/log/postgresql-*.log
```

```
…
… LOG:  database system was interrupted; last known up at 2023-11-05 16:59:03 UTC
… LOG:  invalid checkpoint record
2023-11-05 17:16:52.134 UTC [20177] FATAL:  could not locate required checkpoint
↪   record
2023-11-05 17:16:52.134 UTC [20177] HINT:  If you are restoring from a backup, touch
↪   "/var/lib/pgsql/16/data/recovery.signal" and add required recovery options.
        If you are not restoring from a backup, try removing the file
↪   "/var/lib/pgsql/16/data/backup_label".
        Be careful: removing "/var/lib/pgsql/16/data/backup_label" will result in a
↪   corrupt cluster if restoring from a backup.
```

PostgreSQL ne trouve pas les journaux nécessaires à sa restauration à un état cohérent, le service refuse de démarrer. Il a trouvé un checkpoint dans le fichier `backup_label` créé au début de la sauvegarde, mais aucun checkpoint postérieur dans les journaux (et pour cause).

Les traces contiennent ensuite des suggestions qui peuvent être utiles.

Cependant, un fichier `recovery.signal` ne sert à rien sans `recovery_command`, et nous n'en avons pas encore paramétré ici.

Quant au fichier `backup_label`, le supprimer permettrait peut-être de démarrer l'instance mais celle-ci serait alors dans un état incohérent ! Il y a de bonnes chances que le démarrage s'achève par :

```
PANIC:  could not locate a valid checkpoint record
```

En résumé : la restauration des journaux n'est pas optionnelle !

1.11.2 pg_basebackup : sauvegarde ponctuelle & restauration des journaux suivants

> Remettre en place la copie à froid de l'instance prise précédemment dans `/var/lib/pgsql/16/data.old`. Configurer l'archivage vers un répertoire `/var/lib/pgsql/16/archives`, par exemple avec `rsync`. Configurer la commande de restauration inverse. Démarrer PostgreSQL.

```
sudo systemctl stop postgresql-16  # si nécessaire
```

```
rm -rf /var/lib/pgsql/16/data
cp -rfp /var/lib/pgsql/16/data.old /var/lib/pgsql/16/data
```

Créer le répertoire d'archivage s'il n'existe pas déjà :

```
mkdir /var/lib/pgsql/16/archives
```

Là encore, en production, ce sera pluôt un partage distant. L'utilisateur système **postgres** doit avoir le droit d'y écrire.

L'archivage se définit dans `postgresql.conf` :

```
archive_mode = on
archive_command = 'rsync %p /var/lib/pgsql/16/archives/%f'
```

et on peut y définir aussi tout de suite la commande de restauration :

```
restore_command = 'rsync /var/lib/pgsql/16/archives/%f %p'
```

```
sudo systemctl start postgresql-16
```

> Générer à nouveau de l'activité avec `pgbench`. Vérifier que l'archivage fonctionne.

```
/usr/pgsql-16/bin/pgbench bench -n -P 5 -R 20 -T 720
```

```
ls -lha /var/lib/pgsql/16/archives
```

```
…
-rw-------. 1 postgres postgres 16M Jan  5 18:32 0000000100000000000000BB
-rw-------. 1 postgres postgres 16M Jan  5 18:32 0000000100000000000000BC
-rw-------. 1 postgres postgres 16M Jan  5 18:32 0000000100000000000000BD
…
```

> En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup` au format plain.

```
rm -rf /var/lib/pgsql/16/backups/basebackup
```

```
pg_basebackup -D /var/lib/pgsql/16/backups/basebackup -Fp \
--checkpoint=fast --progress --max-rate=16M
```

```
1586078/1586078 kB (100%), 1/1 tablespace
```

Le répertoire cible devra avoir été vidé.

La taille de la sauvegarde sera bien sûr nettement plus grosse qu'en tar compressé.

> Utiliser `pg_verify_backup` pour contrôler l'intégrité de la sauvegarde.

Si tout va bien, le message sera lapidaire :

```
/usr/pgsql-16/bin/pg_verifybackup  /var/lib/pgsql/16/backups/basebackup
```

```
backup successfully verified
```

S'il y a un problème, des messages de ce genre apparaîtront :

```
pg_verifybackup: error: "global/TEST" is present on disk but not in the manifest
pg_verifybackup: error: "global/2671" is present in the manifest but not on disk
pg_verifybackup: error: "postgresql.conf" has size 29507 on disk but size 29506 in
↪ the manifest
```

> À quoi correspond le fichier finissant par `.backup` dans les archives ?

En effet, parmi les journaux archivés, figure ce fichier :

```
ls -1 /var/lib/pgsql/16/archives
```

```
…
0000000100000000000000BE
0000000100000000000000BE.00003E00.backup
0000000100000000000000BF
…
```

Son contenu correspond au futur `backup_label` :

```
START WAL LOCATION: 0/BE003E00 (file 0000000100000000000000BE)
STOP WAL LOCATION: 0/C864D0F8 (file 0000000100000000000000C8)
CHECKPOINT LOCATION: 0/BE0AB340
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-11-05 18:32:52 UTC
LABEL: pg_basebackup base backup
START TIMELINE: 1
STOP TIME: 2023-11-05 18:34:29 UTC
STOP TIMELINE: 1
```

> Arrêter pgbench et noter la date des dernières données insérées.

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

```
            max
----------------------------
 2023-11-05 18:41:23.068948
```

> Effacer le PGDATA. Restaurer la sauvegarde précédente *sans* les journaux. Configurer la `restore_command`. Créer le fichier `recovery.signal`. Démarrer PostgreSQL.

```
sudo systemctl stop postgresql-16
```

La sauvegarde étant au format *plain*, il s'agit d'une simple copie de fichiers :

```
rm -rf /var/lib/pgsql/16/data/*
rsync -a --exclude 'pg_wal/*' --exclude 'log/*' \
 /var/lib/pgsql/16/backups/basebackup/ \
 /var/lib/pgsql/16/data/
```

Créer le fichier `recovery.signal` :

```
touch /var/lib/pgsql/16/data/recovery.signal
```

Démarrer le service :

```
sudo systemctl start postgresql-16
```

> Vérifier les traces, ainsi que les données restaurées une fois le service démarré.

Les traces sont plus complexes à cause de la restauration depuis les archives :

```
tail -F /var/lib/pgsql/16/data/log/postgresql-*.log
```

```
…
… LOG:  database system was interrupted; last known up at 2023-11-05 18:32:52 UTC
rsync: link_stat "/var/lib/pgsql/16/archives/00000002.history" failed: No such file
 ↪  or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) [sender=3.1.3]
… LOG:  starting archive recovery
… LOG:  restored log file "0000000100000000000000BE" from archive
… LOG:  redo starts at 0/BE003E00
… LOG:  restored log file "0000000100000000000000BF" from archive
… LOG:  restored log file "0000000100000000000000C0" from archive
… LOG:  restored log file "0000000100000000000000C1" from archive
…
… LOG:  restored log file "0000000100000000000000C8" from archive
… LOG:  restored log file "0000000100000000000000C9" from archive
… LOG:  consistent recovery state reached at 0/C864D0F8
… LOG:  database system is ready to accept read-only connections
… LOG:  restored log file "0000000100000000000000CA" from archive
… LOG:  restored log file "0000000100000000000000CB" from archive
…
```

```
… LOG:  restored log file "0000000100000000000000E0" from archive
… LOG:  restored log file "0000000100000000000000E1" from archive
… LOG:  redo in progress, elapsed time: 10.25 s, current LSN: 0/E0FF3438
… LOG:  restored log file "0000000100000000000000E2" from archive
… LOG:  restored log file "0000000100000000000000E3" from archive
…
… LOG:  restored log file "0000000100000000000000EF" from archive
… LOG:  restored log file "0000000100000000000000F0" from archive
rsync: link_stat "/var/lib/pgsql/16/archives/0000000100000000000000F1" failed: No
 ↪  such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
rsync: link_stat "/var/lib/pgsql/16/archives/0000000100000000000000F1" failed: No
 ↪  such file or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
… LOG:  redo done at 0/F0A6C9E0 system usage:
                                          CPU: user: 2.51 s, system: 2.28 s, elapsed:
 ↪  15.77 s
… LOG:  last completed transaction
                                        was at log time 2023-11-05 18:41:23.077219+00
… LOG:  restored log file "0000000100000000000000F0" from archive
rsync: link_stat "/var/lib/pgsql/16/archives/00000002.history" failed: No such file
 ↪  or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
… LOG:  selected new timeline ID: 2
rsync: link_stat "/var/lib/pgsql/16/archives/00000001.history" failed: No such file
 ↪  or directory (2)
rsync error: some files/attrs were not transferred (see previous errors) (code 23)
 ↪  at main.c(1189) …
… LOG:  archive recovery complete
… LOG:  checkpoint starting: end-of-recovery immediate wait
… LOG:  checkpoint complete: wrote 16012 buffers (97.7%); …
… LOG:  database system is ready to accept connections
```

Les messages d'erreur de `rsync` ne sont pas inquiétants : celui-ci ne trouve simplement pas les fichiers demandés à la `restore_command`. PostgreSQL sait ainsi qu'il n'y a pas de fichier `00000002.history` et donc pas de timeline de ce numéro. Il devine aussi qu'il a restauré tous les journaux quand la récupération de l'un d'entre eux échoue.

Les erreurs sur les fichiers `00000001.history` et `00000002.history` sont normales. PostgreSQL cherche à tout hasrd ces fichiers pour voir quel est l'enchaînement des *timelines* et quelle est la dernière.

La progression de la restauration peut être suivie grâce aux différents messages, repris ci-dessous, de démarrage, d'atteinte du point de cohérence, de statut… jusqu'à l'heure exacte de restauration. Enfin, il y a bascule sur une nouvelle *timeline*, et un checkpoint.

```
LOG:  starting archive recovery
LOG:  redo starts at 0/BE003E00
LOG:  consistent recovery state reached at 0/C864D0F8
LOG:  redo in progress, elapsed time: 10.25 s, current LSN: 0/E0FF3438
LOG:  redo done at 0/F0A6C9E0 …
```

```
LOG:  last completed transaction was at log time 2023-11-05 18:41:23.077219+00
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  checkpoint complete:
```

Noter que les journaux portent une nouvelle *timeline* numérotée 2 :

```
ls -l /var/lib/pgsql/16/data/pg_wal/
```

```
…
-rw-------. 1 postgres postgres 16777216 Jan  5 18:43 000000020000000100000023
-rw-------. 1 postgres postgres 16777216 Jan  5 18:43 000000020000000100000024
-rw-------. 1 postgres postgres       42 Jan  5 18:43 00000002.history
drwx------. 2 postgres postgres       35 Jan  5 18:43 archive_status
```

> Vérifier quelles données ont été restaurées.

Cette fois, toutes les données générées après la sauvegarde ont bien été récupérées :

```
psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
```

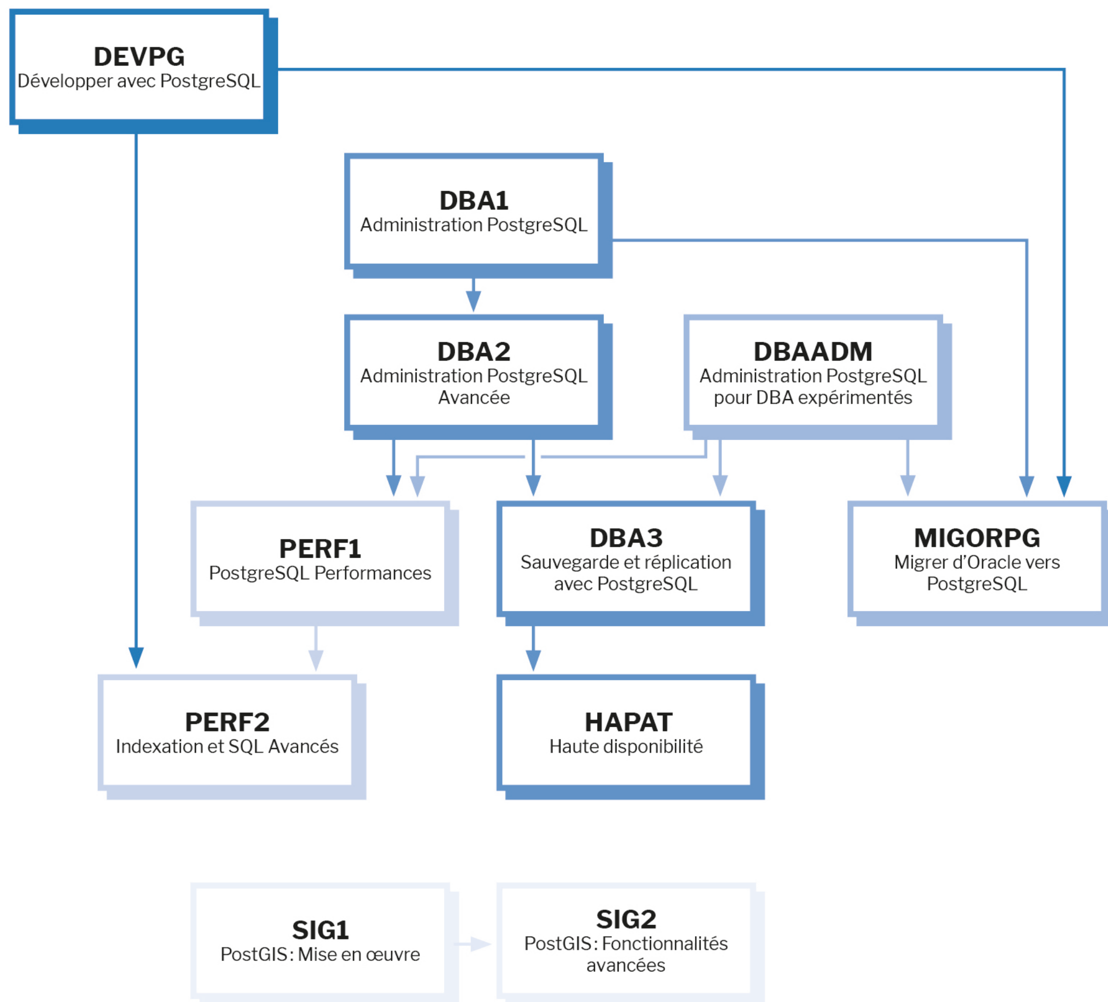
```
            max
----------------------------
 2023-11-05 18:41:23.068948
```

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.