Module I2

Point In Time Recovery

22.09

Dalibo SCOP

https://dalibo.com/formations

Point In Time Recovery

Module I2

TITRE : Point In Time Recovery
SOUS-TITRE : Module I2

REVISION: 22.09
DATE: 02 septembre 2022

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, 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, 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.

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé à :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale : Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

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 !

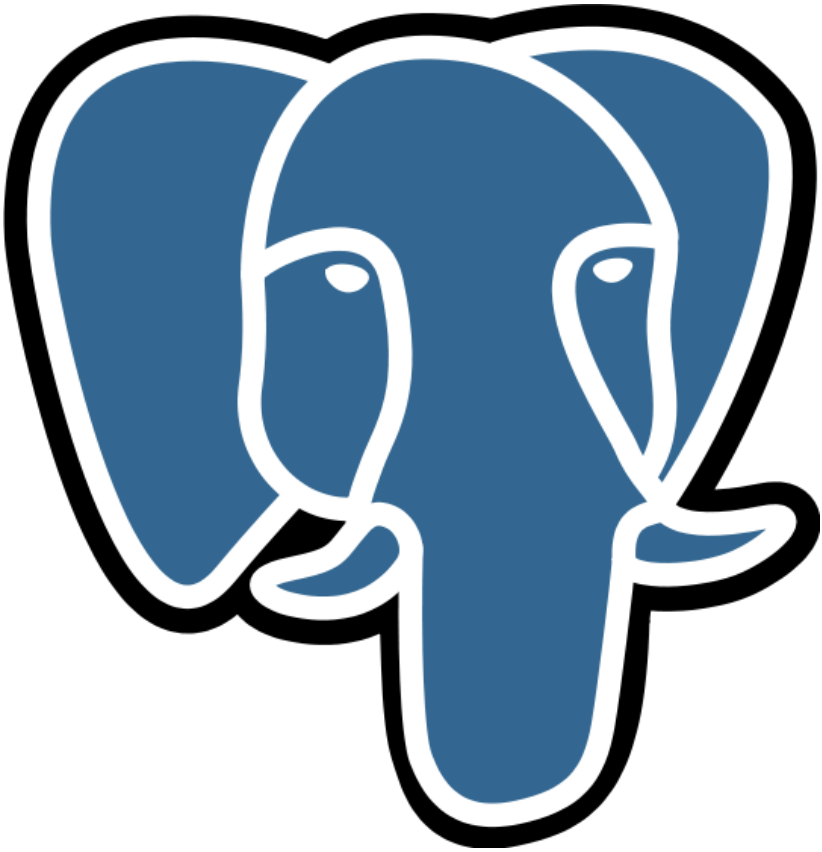
Table des Matières

Licence Creative Commons BY-NC-SA 2.0 FR **5**

1 Sauvegarde physique à chaud et PITR **10**

1.1 Introduction 10
1.2 PITR 11
1.3 Copie physique à chaud ponctuelle avec pg_basebackup 14
1.4 Sauvegarde PITR 16
1.5 Sauvegarde PITR manuelle 26
1.6 Restaurer une sauvegarde PITR 33
1.7 Pour aller plus loin 42
1.8 Conclusion 48
1.9 Quiz 49
1.10 Travaux pratiques 50
1.11 Travaux pratiques (solutions) 54

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 (de manière manuelle ou avec l'outil `pg_receivewal`) à la sauvegarde des fichiers (là aussi, 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.

NB : `pg_receivewal` s'appelait `pg_receivexlog` avant PostgreSQL 10.

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 survient 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 maximale correspondant 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 en cas d'échec d'archivage
  - …avec arrêt si `pg_wal` plein !
- 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
- Copie intégrale,
  - image de la base à la **fin** du backup
- Pas d'incrémental
- 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/
```

`pg_basebackup` est un produit qui a beaucoup évolué dans les dernières versions de PostgreSQL. De plus, le paramétrage par défaut depuis la version 10 le rend immédiatement utilisable.

Il permet de réaliser toute la sauvegarde de la base, à distance, via deux connexions de réplication : une pour les données, une pour les journaux de transactions qui sont générés pendant la copie.

Il est donc 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.

`pg_basebackup` nécessite des connexions de réplication. Par défaut, tout est en place pour cela dans PostgreSQL 10 et suivants pour une connexion en local.

En général, on veut aussi que `pg_basebackup` puisse poser un slot de réplication temporaire :

```
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.

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.

À partir de la v10, 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`.

Lisez bien la documentation de `pg_basebackup`documentation[2] 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[3] .

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`.

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

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 /media/nfs1/archivage
# chown postgres:postgres /media/nfs/archivage
```

1.4.3 PROCESSUS ARCHIVER : CONFIGURATION

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

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

Le premier paramètre à modifier est `wal_level`. Ce paramètre indique le niveau des informations écrites dans les journaux de transactions. Avec un niveau `minimal` (le défaut jusque PostgreSQL 9.6), PostgreSQL peut simplement utiliser les journaux en cas de crash pour rendre les fichiers de données cohérents au redémarrage. Dans le cas d'un archivage, il faut écrire plus d'informations, d'où la nécessité du niveau `replica` (valeur par défaut à partir de PostgreSQL 10).

Avant la version 9.6, il existait deux niveaux intermédiaires pour le paramètre `wal_level` : `archive` et `hot_standby`. Le premier permettait seulement l'archivage, le second permettait en plus d'avoir un serveur secondaire en lecture seule. Ces deux valeurs ont été fusionnées en `replica` avec la version 9.6. Les anciennes valeurs sont toujours acceptées, et remplacées silencieusement par la nouvelle valeur.

Après cela, il faut activer le mode d'archivage en positionnant le paramètre `archive_mode` à `on`. (ou `always` si l'on archive depuis un secondaire).

Enfin, la commande d'archivage s'indique au niveau du paramètre `archive_command`. `archive_command` sert à archiver un seul fichier à chaque appel. PostgreSQL l'appelle une fois pour chaque fichier WAL, dans l'ordre.

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 :

```
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 rapidement 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 d'y placer 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.

Il est donc important de surveiller le processus d'archivage et de faire remonter les problèmes à un opérateur (disque plein, changement de bande, etc.).

Attention, si le code retour de la commande est supérieur à 125, cela va provoquer un redémarrage du processus `archiver`, 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.

Surveiller que la commande fonctionne bien peut se faire simplement en vérifiant la taille du répertoire `pg_wal`. Si ce répertoire commence à grossir fortement, c'est que PostgreSQL n'arrive plus à recycler ses journaux de transactions et ce comportement est un indicateur assez fort d'une commande d'archivage n'arrivant pas à faire son travail. Autre possibilité plus sûre et plus simple : vérifier le nombre de fichiers apparaissant dans le répertoire `pg_wal/archive_status` dont le suffixe est `.ready`. Ces fichiers, de taille nulle, indiquent en permanence quels sont les journaux prêts à être archivés. Théoriquement, leur nombre doit donc rester autour de 0 ou 1. La sonde `check_pgactivity` propose d'ailleurs une action pour faire ce test automatiquement (voir la sonde ready_archives[4] pour plus de détails).

Si l'administrateur souhaite s'assurer qu'un archivage a lieu au moins à une certaine fréquence, il peut configurer un délai maximum avec le paramètre `archive_timeout`. L'impact de ce paramètre est d'archiver des journaux de transactions partiellement remplis. Or, ces fichiers ayant une taille fixe, nous archivons toujours ce nombre d'octets (16 Mo par défaut) par fichier pour un ratio de données utiles beaucoup moins important. La consommation en terme d'espace disque est donc plus importante et le temps de restauration plus long. Ce comportement est désactivé par défaut.

1.4.4 PROCESSUS ARCHIVER : LANCEMENT & SUPERVISION

- Redémarrage de PostgreSQL
  - si modification de `wal_level` et/ou `archive_mode`
- ou rechargement de la configuration
- Supervision avec `pg_stat_archiver` ou `archive_status`

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.

La supervision se fait de trois manières complémentaires. D'abord 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
```

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

```
-[ RECORD 1 ]------+-----------------------------
archived_count     | 156
last_archived_wal  | 000000020000000100000000D9
last_archived_time | 2020-01-17 18:26:03.715946+00
failed_count       | 6
last_failed_wal    | 000000020000000100000000D7
last_failed_time   | 2020-01-17 18:24:24.463038+00
stats_reset        | 2020-01-17 16:08:37.980214+00
```

Pour que cette supervision soit fiable, il faut s'assurer que la commande exécutée renvoie 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.

On peut vérifier les messages d'erreurs 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/000000020000000100000000D7 /opt/pgsql/archives/000000020000000100000000D7
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/000000020000000100000000D7 /opt/pgsql/archives/000000020000000100000000D7
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
```

Enfin, on peut monitorer les fichiers présents dans `pg_wal/archive_status` : ceux en `.ready` indiquent un fichier à archiver, et ne doivent pas s'accumuler :

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

           pg_ls_dir
-------------------------------
 000000020000000100000000DE.done
 000000020000000100000000DF.done
 000000020000000100000000E0.done
 000000020000000100000000E1.ready
 000000020000000100000000E2.ready
 000000020000000100000000E3.ready
 000000020000000100000000E4.ready
 000000020000000100000000E5.ready
 000000020000000100000000E6.ready
 00000002.history.done
```

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` !

1.4.5 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`, 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 au prix d'une certaine latence.

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 v13 :

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

```
Usage :
  pg_receivewal [OPTION]...

Options :
  -D, --directory=RÉP          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=0-9           compresse la sortie tar avec le niveau de
                               compression indiqué
  -?, --help                   affiche cette aide puis quitte

Options de connexion :
  -d, --dbname=CONNSTR         chaîne de connexion
  -h, --host=NOMHÔTE           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=NOM           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 arriver automatiquement)

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

Rapporter les bogues à <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
```

1.4.6 PG_RECEIVEWAL - CONFIGURATION SERVEUR

- `postgresql.conf` (si < v10) :

```
max_wal_senders = 10
max_replication_slots = 10
```

- `pg_hba.conf` :

```
host  replication  repli_user  192.168.0.0/24  md5
```

- 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 à partir de PostgreSQL 10, mais 0 sur les versions précédentes, et il faudra les modifier.

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.

1.4.7 PG_RECEIVEWAL - REDÉMARRAGE DU SERVEUR

- Redémarrage de PostgreSQL
- Slot de réplication

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

Pour que les modifications soient prises en compte, nous devons redémarrer le serveur.

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`.

1.4.8 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

Une fois le serveur PostgreSQL redémarré, 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 000000010000000000000000E*
-rwx------  1 postgres postgres  16MB juil. 27 000000010000000000000000F*
-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.

1.4.9 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 (les quelques dernières transactions)

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 a un

`pg_stop_backup`), 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 être écrit et installé. Cette méthode nécessite donc une configuration plus importante du serveur PostgreSQL. Par contre, elle a le gros avantage de ne perdre pratiquement aucune transaction. 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 :
  - `pg_start_backup()`
  - copie des fichiers par outil externe
  - `pg_stop_backup()`
- Exclusive : simple… & obsolète !
- 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 `pg_start_backup()` ;
- la copie elle-même par divers outils externes (PostgreSQL ne s'en occupe pas) ;
- l'appel à `pg_stop_backup()`.

La sauvegarde exclusive est la plus simple, et le choix par défaut. Il ne peut y en avoir qu'une à la fois. Elle ne fonctionne que depuis un primaire.

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.

Les deux sauvegardes diffèrent par les paramètres envoyés aux fonctions et les fichiers générés dans le PGDATA ou parmi les journaux de transactions.

Dans les deux cas, l'utilisateur ne verra aucun impact (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 sauvegarde exclusive fonctionne encore mais est déclarée obsolète, et la fonctionnalité disparaîtra dans une version majeure prochaine. Il est donc conseillé de créer les nouveaux scripts avec des sauvegardes concurrentes.

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…

1.5.1 SAUVEGARDE MANUELLE - 1/3 : PG_START_BACKUP

`SELECT pg_start_backup (`

- `un_label` : texte
- `fast` : forcer un checkpoint ?
- `exclusive` : sauvegarde exclusive
  - `backup_label`, `tablespace_map`

`)`

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

Le troisième argument permet de demander une sauvegarde exclusive.

Si la sauvegarde est exclusive, l'appel à `pg_start_backup()` va alors créer un fichier `backup_label` dans le répertoire des données de PostgreSQL. Il contient le journal de transactions et l'emplacement actuel dans le journal de transactions du *checkpoint* ainsi que le label précisé en premier argument de la procédure stockée. Ce label permet d'identifier l'opération de sauvegarde. Il empêche l'exécution de deux sauvegardes PITR en parallèle. Voici un exemple de ce fichier `backup_label` en version 13 :

```
$ cat $PGDATA/backup_label
START WAL LOCATION: 0/11000028 (file 000000010000000000000011)
```

```
CHECKPOINT LOCATION: 0/11000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2019-11-25 17:59:29 CET
LABEL: backup_full_2019_11-25
START TIMELINE: 1
```

Il faut savoir qu'en cas de crash pendant une sauvegarde exclusive, la présence d'un `backup_label` peut poser des problèmes : le serveur croit à une restauration depuis une sauvegarde et ne redémarre donc pas. Le message d'erreur invitant à effacer `backup_label` est cependant explicite. C'est un argument pour éviter d'utiliser les sauvegardes exclusives. (Voir les détails dans ce mail de Robert Haas sur pgsql-hackers[5] .)

Un fichier `tablespace_map` dans PGDATA est aussi créé avec les chemins des tablespaces :

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

La sauvegarde sera concurrente si le troisième paramètre à `pg_start_backup()` est à `false`. Les fichiers `backup_label` et `tablespace_map` ne sont alors pas créés.

Les contraintes des sauvegardes en parallèle sont importantes. En effet, la session qui exécute la commande `pg_start_backup()` doit être la même que celle qui exécutera plus tard `pg_stop_backup()`. Si la connexion venait à être interrompue entre-temps, alors la sauvegarde doit être considérée comme invalide. Comme il n'y a plus de fichier `backup_label` c'est la commande `pg_stop_backup()` qui renverra les informations qui s'y trouvaient, comme nous le verrons plus tard.

Si vos scripts doivent gérer la sauvegarde concurrente, il vous faudra donc récupérer et conserver vous-même les informations renvoyées par la commande de fin de sauvegarde.

La sauvegarde PITR devient donc plus complexe au fil des versions, et il est donc recommandé d'utiliser plutôt `pg_basebackup` ou des outils supportant ces fonctionnalités (pitrery, pgBackRest...).

---

[5] https://www.postgresql.org/message-id/CA+TgmoaGvpybE=xvJeg9Jc92c-9ikeVz3k-_Hg9=mdG05u=e=g@mail.gmail.com

1.5.2 SAUVEGARDE MANUELLE - 2/3 : COPIE DES FICHIERS

- 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.

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. À noter : l'outil tar (ainsi que d'autres issus du projet GNU) est disponible nativement à travers le projet unxutils[6] .

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[7]) :

- `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 ;

[6] http://unxutils.sourceforge.net/
[7] https://docs.postgresql.fr/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP

- `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.

1.5.3 SAUVEGARDE MANUELLE - 3/3 : PG_STOP_BACKUP

Ne pas oublier !!

`SELECT * FROM pg_stop_backup (`

- `false` : si concurrente
  - fichier `.backup`
- `true` : attente de l'archivage

`)`

La dernière étape correspond à l'exécution de la procédure stockée `SELECT * FROM pg_stop_backup (true|false)`, le `false` étant alors obligatoire si la sauvegarde était concurrente.

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

PostgreSQL va :

- 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 immédiat : si l'archivage est en place, `pg_stop_backup()` ne rendra pas la main (par défaut) tant que ce dernier journal n'aura pas été archivé avec succès ;
- supprimer ou renommer les fichiers `backup_label` et `tablespace_map`.

En cas de sauvegarde concurrente, il faut passer à la fonction `pg_stop_backup` un premier argument booléen à `false`. Dans ce cas, la fonction renvoie :

- le lsn de fin de backup ;
- l'équivalent du contenu du fichier `backup_label` ;
- l'équivalent du contenu du fichier `tablespace_map`.

Ce contenu doit être conservé : lors de la restauration, il servira à recréer les fichiers `backup_label` ou `tablespace_map` qui seront stockés dans le PGDATA.

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

NOTICE:  all required WAL segments have been archived
-[ RECORD 1 ]----------------------------------------------------------
lsn        | 22/2FE5C788
labelfile  | START WAL LOCATION: 22/2B000028 (file 000000010000002200000002B)+
           | CHECKPOINT LOCATION: 22/2B000060                             +
           | BACKUP METHOD: streamed                                      +
           | BACKUP FROM: master                                          +
           | START TIME: 2019-12-16 13:53:41 CET                         +
           | LABEL: rr                                                    +
           | START TIMELINE: 1                                           +
           |
spcmapfile | 134141 /tbl/froid                                           +
           | 134152 /tbl/quota                                           +
           |
```

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

```
# cat /var/lib/postgresql/12/main/pg_wal/000000010000002200000002B.00000028.backup

START WAL LOCATION: 22/2B000028 (file 000000010000002200000002B)
STOP WAL LOCATION: 22/2FE5C788 (file 000000010000002200000002F)
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
```

À partir du moment où `pg_stop_backup` 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 COMPLÈTE À CHAUD : PG_BASEBACKUP

- Réalise les différentes étapes d'une sauvegarde
  - via une connexion de réplication
- Crée un fichier manifeste (v13+)
- Backup de base **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.

Le fichier manifeste permet de vérifier une sauvegarde grâce à l'outil `pg_verifybackup`.

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 l'ancienne 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)

- Fichier de configuration
  - jusque v11 : `recovery.conf`
  - v12 et au-delà : `postgresql.[auto.]conf` + `recovery.signal`
- Comment restaurer :
  - `restore_command = '... une commande ...'`

Jusqu'en version 11 incluse, la restauration se configure dans un fichier spécifique, appelé `recovery.conf`, impérativement dans le PGDATA.

À partir de la 12, on utilise directement `postgresql.conf`, ou un fichier inclus, ou `postgresql.auto.conf`. Par contre, il faut créer un fichier vide `recovery.signal`.

Ce sont ces fichiers `recovery.signal` ou `recovery.conf` qui permettent à PostgreSQL de savoir qu'il doit se lancer dans une restauration, et n'a pas simplement subi un arrêt brutal

(auquel cas il ne restaurerait que les journaux en place).

Si vous êtes sur une version antérieure à la version 12, vous pouvez vous inspirer du fichier exemple fourni avec PostgreSQL. Pour une version 10 par exemple, sur Red Hat et dérivés, c'est `/usr/pgsql-10/share/recovery.conf.sample`.
Sur Debian et dérivés, c'est `/usr/share/postgresql/10/recovery.conf.sample`. Il contient certains paramètres liés à la mise en place d'un serveur secondaire (*standby*) inutiles ici. Sinon, les paramètres sont dans les différentes parties du `postgresql.conf`.

Le paramètre essentiel est `restore_command`. Il est le pendant du paramètre `archive_command` pour l'archivage. Cette commande est souvent fournie par l'outil de sauvegarde si vous en utilisez un. Si nous poursuivons notre exemple, ce paramètre pourrait être :

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

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 l'épuisement de tous les journaux disponibles.

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`
- 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*[8]), grâce au paramètre `recovery_target_lsn`, 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* (avec `recovery_target_timeline`, en général à `latest`) peut être utile.

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`.

Il est possible de demander à la restauration de s'arrêter une fois arrivée au stade voulu avec `recovery_target_action = pause` (ou, jusqu'en 9.4 incluse, `pause_at_recovery_target = true`). 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 néglige les opérations contenues dans les WALs que l'on n'a pas souhaité restaurer, puis le serveur s'ouvre en écriture sur une nouvelle timeline.

Attention : jusque PostgreSQL 12 inclus, si un `recovery_target` est spécifié mais n'est toujours *pas* atteint à la fin du rejeu des archives, alors le mode *recovery* se termine 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 !

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

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[9] .

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 du `pg_stop_backup`. 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, ou que le bloc de journal lu soit incohérent (ce qui 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). Par défaut en v12 il vérifiera 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"…
```

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

```
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 "000000010000000010000003F"…
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
```

À partir de la version 12, le fichier `recovery.signal` est effacé.

Avant la v12, si un fichier `recovery.conf` est présent, il sera renommé en `recovery.done`.

Ne jamais supprimer ou renommer manuellement un `recovery.conf` !

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

1.6.6 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` (v12) ou `current` (!) (<v12)

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/
000000010000000100000003C
000000010000000100000003D
000000010000000100000003E
000000010000000100000003F
000000010000000100000040
00000002.history
000000020000000100000003F
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 *history*, nommés d'après le numéro hexadécimal sur huit caractères de la *timeline* et le suffixe `.history`, et archivés avec les fichiers WAL. En partant de la *timeline* qu'elle quitte, l'instance restaure les fichiers *history* des *timelines* suivantes pour choisir la première disponible, et 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`.

Par défaut, jusqu'en version 11 comprise, sa valeur est `current` et la restauration se fait dans la même *timeline* que le *base backup*. Si entre-temps il y a eu une bascule ou une précédente restauration, la nouvelle timeline ne sera pas automatiquement suivie !

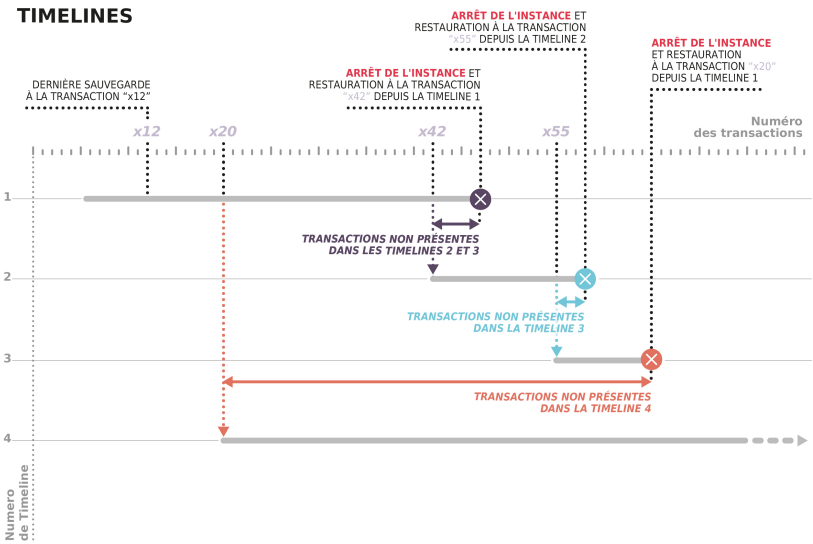
À partir de la version 12, `recovery_target_timeline` est par défaut à `latest` et la restauration suit les changements de *timeline* depuis le moment du *base backup*.

Pour choisir une autre *timeline*, il faut donner le numéro hexadécimal de la *timeline* cible comme valeur du paramètre `recovery_target_timeline`. Cela permet d'effectuer

plusieurs restaurations successives à partir du même *base backup* et d'archiver au même endroit sans mélanger les journaux.

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

1.6.7 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.

1.7 POUR ALLER PLUS LOIN

- Gagner en place
  - ...en compressant les journaux de transactions
- Les outils dédiés à la sauvegarde

1.7.1 COMPRESSER LES JOURNAUX DE TRANSACTIONS

- Objectif : éviter de consommer trop de place disque
- Outils de compression standards : `gzip`, `bzip2`, `lzma`
  - attention à ne pas ralentir l'archivage
- `wal_compression`

L'un des problèmes de la sauvegarde PITR est la place prise sur disque par les journaux de transactions. Avec un journal généré toutes les 5 minutes, cela représente 16 Mo (par défaut) toutes les 5 minutes, soit 192 Mo par heure, ou 5 Go par jour. Il n'est pas forcément possible de conserver autant de journaux. Une solution est la compression à la volée et il existe deux types de compression.

La méthode la plus simple et la plus sûre pour les données est une compression non destructive, comme celle proposée par les outils `gzip`, `bzip2`, `lzma`, etc. L'algorithme peut être imposé ou inclus dans l'outil PITR choisi. La compression peut ne pas être très intéressante en terme d'espace disque gagné. Néanmoins, un fichier de 16 Mo aura généralement une taille compressée comprise entre 3 et 6 Mo. Attention par ailleurs au temps de compression des journaux, qui peut entraîner un retard conséquent de l'archivage par rapport à l'écriture des journaux en cas d'écritures lourdes : une compression élevée mais lente peut être dangereuse.

Noter que la compression des journaux à la source existe aussi (paramètre `wal_compression`, désactivé par défaut), qui s'effectue au niveau de la page, avec un coût en CPU à l'écriture des journaux.

1.7.2 OUTILS DE SAUVEGARDE PITR DÉDIÉS

- Se faciliter la vie avec différents outils
  - pgBackRest
  - barman
  - pitrery (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 succintement 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.

1.7.3 PGBACKREST

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

pgBackRest[10] 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 :

[10] https://pgbackrest.org/

- 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 notamment),
- le support intégré de dépôts S3 ou Azure ;
- 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.

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

Pour la supervision de l'outil, une sonde Nagios est fournie par un des développeurs : check_pgbackrest[11] .

1.7.4 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 ;

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

- `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 choisit d'utiliser `pg_receivewal`.

Il supporte aussi les dépôts S3 ou blob Azure.

Site web de barman[12]

1.7.5 PITRERY

- Gère la sauvegarde et la restauration
  - mode push
  - mono-serveur
- Multi-commandes
  - `archive_wal`
  - `pitrery`
  - `restore_wal`
- Projet en fin de vie, fin du support en version 14.

pitrery a été créé par la société Dalibo. Il met l'accent sur la simplicité de sauvegarde et la restauration de la base. Cet outil s'appuie sur des fichiers de configuration, un par serveur de sauvegarde, qui permettent de définir la destination de l'archivage, la destination des sauvegardes ainsi que la politique de rétention à utiliser.

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.

pitrery propose trois exécutables :

- `archive_wal` est appelé par l'`archive_command` et gère l'archivage et la compression des journaux de transactions ;
- `pitrery` gère les sauvegardes et les restaurations ;
- `restore_wal` est appelé par `restore_command` et restaure des journaux archivés par `archive_wal`.

[12] https://www.pgbarman.org/

Note : les versions précédant la version 3 de pitrery utilisent les scripts `archive_xlog` et `restore_xlog`.

Lorsque l'archivage est fonctionnel, la commande `pitrery` peut être utilisée pour réaliser une sauvegarde :

```
$ pitrery backup
INFO: preparing directories in 10.100.0.16:/opt/backups/prod
INFO: listing tablespaces
INFO: starting the backup process
INFO: backing up PGDATA with tar
INFO: archiving /home/postgres/postgresql-9.0.4/data
INFO: backup of PGDATA successful
INFO: backing up tablespace "ts2" with tar
INFO: archiving /home/postgres/postgresql-9.0.4/ts2
INFO: backup of tablespace "ts2" successful
INFO: stopping the backup process
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
INFO: copying the backup history file
INFO: copying the tablespaces list
INFO: backup directory is 10.100.0.16:/opt/backups/prod/2013.08.28-11.16.30
INFO: done
```

Il est possible d'obtenir la liste des sauvegardes en ligne :

```
$ pitrery list
List of backups on 10.100.0.16:

Directory:
  /usr/data/pitrery/backups/pitr13/2013.05.31_11.44.02
Minimum recovery target time:
  2013-05-31 11:44:02 CEST
Tablespaces:

Directory:
  /usr/data/pitrery/backups/pitr13/2013.05.31_11.49.37
Minimum recovery target time:
  2013-05-31 11:49:37 CEST
Tablespaces:
  ts1 /opt/postgres/ts1 (24576)
```

pitrery gère également la politique de rétention des sauvegardes. Une commande de purge permet de réaliser la purge des sauvegardes en s'appuyant sur la configuration de la rétention des sauvegardes :

```
$ pitrery purge
INFO: searching backups
```

```
INFO: purging /home/postgres/backups/prod/2011.08.17-11.16.30
INFO: purging WAL files older than 000000020000000000000060
INFO: 75 old WAL file(s) removed
INFO: done
```

pitrery permet de restaurer une sauvegarde et de préparer la configuration de restauration :

```
$ pitrery -c prod restore -d '2013-06-01 13:00:00 +0200'
INFO: searching backup directory
INFO: searching for tablespaces information
INFO:
INFO: backup directory:
INFO:   /opt/postgres/pitr/prod/2013.06.01_12.15.38
INFO:
INFO: destinations directories:
INFO:   PGDATA -> /opt/postgres/data
INFO:   tablespace "ts1" -> /opt/postgres/ts1 (relocated: no)
INFO:   tablespace "ts2" -> /opt/postgres/ts2 (relocated: no)
INFO:
INFO: recovery configuration:
INFO:   target owner of the restored files: postgres
INFO:   restore_command = 'restore_xlog -L -d /opt/postgres/archives %f %p'
INFO:   recovery_target_time = '2013-06-01 13:00:00 +0200'
INFO:
INFO: checking if /opt/postgres/data is empty
INFO: checking if /opt/postgres/ts1 is empty
INFO: checking if /opt/postgres/ts2 is empty
INFO: extracting PGDATA to /opt/postgres/data
INFO: extracting tablespace "ts1" to /opt/postgres/ts1
INFO: extracting tablespace "ts2" to /opt/postgres/ts2
INFO: preparing pg_wal directory
INFO: preparing recovery.conf file
INFO: done
INFO:
INFO: please check directories and recovery.conf before starting the cluster
INFO: and do not forget to update the configuration of pitrery if needed
INFO:
```

Il ne restera plus qu'à redémarrer le serveur et surveiller les journaux applicatifs pour vérifier qu'aucune erreur ne se produit au cours de la restauration.

pitrery est capable de vérifier qu'il possède bien l'intégralité des journaux nécessaires à la restauration depuis les plus anciennes sauvegardes complètes, et que des backups assez récents sont bien là :

```
$ pitrery  check -A -B -g 1d
```

```
2019-03-04 09:12:25 CET INFO: checking local backups in /opt/postgres/archives
2019-03-04 09:12:25 CET INFO: newest backup age: 5d 22h 54min 53s
2019-03-04 09:12:25 CET INFO: number of backups: 3
2019-03-04 09:12:25 CET ERROR: backups are too old
2019-03-04 09:12:25 CET INFO: checking local archives in
                              /opt/postgres/archives/archived_xlog
2019-03-04 09:12:25 CET INFO: oldest backup is:
                              /opt/postgres/archives/2019.02.18_23.39.33
2019-03-04 09:12:25 CET INFO: start wal file is: 000000020000011500000003C
2019-03-04 09:12:25 CET INFO: listing WAL files
2019-03-04 09:12:25 CET INFO: looking for WAL segment size
2019-03-04 09:12:25 CET INFO: WAL segment size: 32MB
2019-03-04 09:12:25 CET INFO: WAL segments per log: 127
2019-03-04 09:12:25 CET INFO: first WAL file checked is: 000000020000011500000003C.gz
2019-03-04 09:12:25 CET INFO: start WAL file found
2019-03-04 09:12:27 CET INFO: looking for 000000020000011800000000
                              (checked 324/1004, 162 segs/s)
2019-03-04 09:12:30 CET INFO: looking for 000000020000011C00000000
                              (checked 836/1004, 167 segs/s)
2019-03-04 09:12:31 CET INFO: last WAL file checked is: 000000020000011D00000023.gz
2019-03-04 09:12:31 CET INFO: all archived WAL files found
```

Pour faciliter la supervision, `pitrery check -B -A -n` permet à pitrery de se comporter comme sa propre sonde Nagios.

Site Web de pitrery[13] .

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).

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

1.8.1 QUESTIONS

- N'hésitez pas, c'est le moment !

1.9 QUIZ

- https://dali.bo/i2_quiz

1.10 TRAVAUX PRATIQUES

Pour simplifier les choses, merci de ne pas créer de tablespaces. La solution fournie n'est valable que dans ce cas précis. Dans le cas contraire, il vous faudra des sauvegardes séparées de chaque tablespace.

1.10.1 SAUVEGARDE MANUELLE

But : Mettre en place l'archivage et faire une sauvegarde physique à chaud manuelle avec `pg_start_backup` et `pg_stop_backup`.

Mettre en place l'archivage des journaux de transactions vers `/opt/pgsql/archives` avec `rsync`.

Générer de l'activité avec `pgbench` et laisser tourner en arrière-plan.

Vérifier que les journaux de transactions arrivent bien dans le répertoire d'archivage.

Indiquer à PostgreSQL qu'une sauvegarde exclusive va être lancée.
Sauvegarder l'instance à l'aide d'un utilitaire d'archivage (`tar` par exemple) vers `/opt/pgsql/backups/`. Penser à exclure le contenu des répertoires `pg_wal` et `log`.
À la fin de la sauvegarde, relever la dernière valeur de `pgbench_history.mtime`.
Ne pas oublier la dernière opération après la sauvegarde.

Attendre qu'au moins un nouveau journal de transactions soit archivé, puis arrêter l'activité.

1.10.2 RESTAURATION

But : Restaurer une sauvegarde physique.

> Arrêter PostgreSQL.

> Renommer le répertoire `/var/lib/pgsql/14/data` en `/var/lib/pgsql/14/data.old`.

> Comparer le dernier journal de l'instance arrêtée et le dernier archivé, et le déplacer dans le répertoire d'archivage des journaux de transactions au besoin.

> Restaurer le PGDATA de l'instance en utilisant la sauvegarde à chaud : restaurer l'archive, vérifier le contenu de `backup_label`.

> Effacer les journaux éventuellement présents dans ce nouveau PGDATA.

> Créer les fichiers nécessaires à la restauration et renseigner `restore_command`.

> Afficher en continu les traces dans une fenêtre avec `tail -F`.

> Relancer l'instance et attendre la fin de la restauration.
> Vérifier que les journaux sont réappliqués puis que le point de cohérence a été atteint.
> Que se passe-t-il quand le dernier journal a été restauré ?

> Quels sont les fichiers, disparus, modifiés ?

> Vérifier que la connexion peut se faire.
> Générer des journaux : quel est leur nom ?

> Récupérer la dernière valeur de `pgbench_history.mtime`. Vérifier qu'elle est supérieure à celle relevée à la fin de la sauvegarde.

1.10.3 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` : activer l'archivage, autoriser des connexions en streaming.

> Insérer des données et générer de l'activité avec `pgbench`.
> En parallèle, sauvegarder l'instance avec `pg_basebackup` au format tar, sans oublier les journaux, et avec l'option `--max-rate=16M` pour ralentir la sauvegarde.

> Tout au long de l'exécution de la sauvegarde, surveiller l'évolution de l'activité sur la table `pgbench_history`.

> Une fois la sauvegarde terminée, vérifier son état ainsi que la dernière donnée modifiée dans `pgbench_history`.

> Arrêter l'instance.
> Par sécurité, faites une copie à froid des données (par exemple avec `cp -rfp`).
> Vider le répertoire.
> Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.

> 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.

1.10.4 PG_BASEBACKUP : SAUVEGARDE PONCTUELLE & RESTAURATION

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

> Remettre en place la copie de l'instance prise précédemment.
> Configurer l'archivage.

> Générer à nouveau de l'activité avec `pgbench`.
> En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup`.

> Vérifier que des archives sont bien générées.

> Effacer le PGDATA.
> Restaurer la sauvegarde, configurer la `restore_command` et créer le fichier `recovery.signal`.

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

1.11 TRAVAUX PRATIQUES (SOLUTIONS)

1.11.1 SAUVEGARDE MANUELLE

> Mettre en place l'archivage des journaux de transactions vers `/opt/pgsql/archives` avec `rsync`.

D'abord créer le répertoire `/opt/pgsql/archives` avec les droits nécessaire pour l'utilisateur système **postgres** :

```
# mkdir -p /opt/pgsql/archives
# chown -R postgres /opt/pgsql
```

Pour mettre en place l'archivage des journaux de transactions dans `/opt/pgsql/archives`, il faut positionner la variable `archive_command` comme suit dans le fichier de configuration `postgresql.conf` :

```
archive_command = 'rsync %p /opt/pgsql/archives/%f'
```

Positionner le paramètre `archive_mode` à `on`.

Un redémarrage est nécessaire si le paramètre `archive_mode` a été modifié :

```
# systemctl restart postgresql-14
```

Si seul `archive_command` a été modifié, `systemctl reload postgresql-14` suffira.

> Générer de l'activité avec `pgbench` et laisser tourner en arrière-plan.

Si la base `pgbench` n'existe pas déjà, on peut créer une base d'environ 1 Go ainsi :

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 100 pgbench
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data...
100000 of 10000000 tuples (1%) done (elapsed 0.09 s, remaining 9.39 s)
200000 of 10000000 tuples (2%) done (elapsed 0.36 s, remaining 17.46 s)
…
10000000 of 10000000 tuples (100%) done (elapsed 28.57 s, remaining 0.00 s)
vacuuming...
```

```
creating primary keys...
done.
```

L'activité peut être générée par exemple ainsi pendant 30 min :

```
/usr/pgsql-14/bin/pgbench -n -T1800 -c 10 -j2 pgbench
```

> Vérifier que les journaux de transactions arrivent bien dans le répertoire d'archivage.

La quantité de journaux générés dépend de la machine et des disques. Ils doivent se retrouver dans `/opt/pgsql/archives/` :

```
$ ls -l /opt/pgsql/archives
total 540672
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000BE
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000BF
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C0
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C1
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C2
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C3
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C4
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C5
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C6
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C7
-rw-------. 1 postgres postgres 16777216 17 janv. 14:47 0000000100000000000000C8
…
```

> Indiquer à PostgreSQL qu'une sauvegarde exclusive va être lancée.
> Sauvegarder l'instance à l'aide d'un utilitaire d'archivage (`tar` par exemple) vers `/opt/pgsql/backups/`. Penser à exclure le contenu des répertoires `pg_wal` et `log`.
> À la fin de la sauvegarde, relever la dernière valeur de `pgbench_history.mtime`.
> Ne pas oublier la dernière opération après la sauvegarde.

Indiquer à PostgreSQL que la sauvegarde exclusive va démarrer :

```
SELECT pg_start_backup('backup '||current_timestamp, true);
```

Sauvegarder les fichiers de l'instance :

```
$ cd /var/lib/pgsql/14/data/
$ mkdir -p /opt/pgsql/backups
```

```
$ tar -cvhz . -f /opt/pgsql/backups/backup_$(date +%F).tgz \
  --exclude='pg_wal/*' --exclude='log/*'

./
./pg_wal/
./global/
./global/pg_control
./global/1262
./global/2964
./global/1213
./global/1136
…
./pg_ident.conf
./log/
./log/postgresql-Fri.log
./postmaster.opts
./postgresql.conf
./postmaster.pid
./current_logfiles
./backup_label
```

Noter que l'erreur suivante est sans conséquence, les fichiers peuvent être modifiés pendant une copie à chaud :

```
./base/16384/16397
tar: ./base/16384/16397: file changed as we read it
```

Enfin, il ne faut surtout pas oublier d'indiquer à PostgreSQL la fin de la sauvegarde :

```
SELECT pg_stop_backup();

NOTICE:  all required WAL segments have been archived
 pg_stop_backup
----------------
 1/32BFDD88
```

On repère la dernière valeur :

```
$ psql pgbench
pgbench=# SELECT max(mtime) FROM pgbench_history ;

            max
---------------------------
  2020-01-17 14:58:28.423145
```

> Attendre qu'au moins un nouveau journal de transactions soit archivé, puis arrêter l'activité.

Forcer au besoin un changement de journal avec :

```
SELECT pg_switch_wal();
```

1.11.2 RESTAURATION

> Arrêter PostgreSQL.

En tant qu'utilisateur root :

```
# systemctl stop postgresql-14
```

Alternativement, on peut être plus brutal et faire un `kill -9` sur le `postmaster`.

> Renommer le répertoire `/var/lib/pgsql/14/data` en `/var/lib/pgsql/14/data.old`.

```
$ mv /var/lib/pgsql/14/data /var/lib/pgsql/14/data.old
```

> Comparer le dernier journal de l'instance arrêtée et le dernier archivé, et le déplacer dans le répertoire d'archivage des journaux de transactions au besoin.

Avant une restauration, il faut toujours chercher à sauver les toutes dernières transactions. Ici le dernier journal archivé est :

```
$ ls -lrt /opt/pgsql/archives/ | tail -n1
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 00000001000000010000003E
```

On recherche le dernier modifié :

```
$ ls -lr /var/lib/pgsql/14/data.old/pg_wal
total 1048592
drwx------. 2 postgres postgres     8192 17 janv. 14:59 archive_status
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 00000001000000010000007E
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 00000001000000010000007D
…
-rw-------. 1 postgres postgres 16777216 17 janv. 14:53 000000010000000100000043
-rw-------. 1 postgres postgres 16777216 17 janv. 14:52 000000010000000100000042
-rw-------. 1 postgres postgres 16777216 17 janv. 14:52 000000010000000100000041
-rw-------. 1 postgres postgres 16777216 17 janv. 14:54 000000010000000100000040
-rw-------. 1 postgres postgres 16777216 17 janv. 14:59 00000001000000010000003F
-rw-------. 1 postgres postgres      358 17 janv. 14:58 00000001000000000000000FA.00370610.backup
```

Attention ! Selon l'activité il peut y avoir de nombreux journaux recyclés, mais ne contenant pas de données intéressantes. Il faut regarder la date de modification.

Ci-dessus le dernier journal intéressant est bien le `00000001000000010000003F`, ceux à partir de `000000010000000100000040` sont d'anciens journaux modifiés précédemment et renommés.

PostgreSQL n'a pas eu le temps de terminer et d'archiver ce journal, il faut le récupérer manuellement :

```
$ cp /var/lib/pgsql/14/data.old/pg_wal/00000001000000010000003F  /opt/pgsql/archives/
```

> Restaurer le PGDATA de l'instance en utilisant la sauvegarde à chaud : restaurer l'archive, vérifier le contenu de `backup_label`.

Restaurer l'archive de la dernière sauvegarde :

```
$ cd /var/lib/pgsql/14
$ mkdir data
$ cd data
$ tar xzvf /opt/pgsql/backups/backup_$(date +%F).tgz
…
```

Le `backup_label` a été créé le temps de la sauvegarde par `pg_start_backup` (ce ne serait pas le cas dans une sauvegarde concurrente, PostgreSQL utilise d'autres moyens) et est donc inclus dans la sauvegarde.

```
$ cat backup_label
START WAL LOCATION: 0/FA370610 (file 000000010000000000000FA)
CHECKPOINT LOCATION: 0/FA604098
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2020-01-17 14:52:51 UTC
LABEL: backup 2020-01-17 14:52:48.456761+00
START TIMELINE: 1
```

> Effacer les journaux éventuellement présents dans ce nouveau PGDATA.

Effacer les anciens journaux de l'archive que l'on vient d'extraire, si on avait oublié de les extraire. On peut en profiter pour effacer certains fichiers comme `postmaster.pid` :

```
$ rm -f /var/lib/pgsql/14/data/pg_wal/00*   postmaster.pid
```

> Créer les fichiers nécessaires à la restauration et renseigner `restore_command`.

À partir de la v12 : ajouter la commande de récupération des journaux dans `postgresql.auto.conf` :

```
restore_command = 'cp /opt/pgsql/archives/%f %p'
```

Toujours à partir de la v12 : créer le fichier `recovery.signal` :

```
$ touch /var/lib/pgsql/14/data/recovery.signal
```

Jusqu'en v11 : créer le fichier `recovery.conf` dans le PGDATA, et y préciser `restore_command`. Ce fichier peut aussi être copié depuis un fichier d'exemple fourni avec PostgreSQL (sur Red Hat/CentOS/Rocky Linux : `/usr/pgsql-11/share/recovery.conf.sample`), qui liste toutes les options possibles.

> Afficher en continu les traces dans une fenêtre avec `tail -F`.

Le nom du fichier peut différer suivant le jour de la semaine et le paramétrage. Noter aussi qu'il n'existe pas si `log/*` a bien été exclu du `tar`. `tail -F` l'affichera dès qu'il apparaîtra :

```
$ tail -F /var/lib/pgsql/14/data/log/postgresql-Thu.log
```

> Relancer l'instance et attendre la fin de la restauration.
> Vérifier que les journaux sont réappliqués puis que le point de cohérence a été atteint.
> Que se passe-t-il quand le dernier journal a été restauré ?

Redémarrer le serveur PostgreSQL :

```
# systemctl start postgresql-14
```

La trace indique que PostgreSQL a repéré qu'il était interrompu, et il commence à restaurer les journaux :

```
2020-01-17 16:08:06.373 UTC [15221] LOG:  database system was interrupted;
                                    last known up at 2020-01-17 14:52:51 UTC
cp: cannot stat ‘/opt/pgsql/archives/00000002.history’: No such file or directory
2020-01-17 16:08:06.457 UTC [15221] LOG: starting archive recovery
2020-01-17 16:08:06.524 UTC [15221] LOG: restored log file "0000000100000000000000FA"…
2020-01-17 16:08:06.657 UTC [15221] LOG: redo starts at 0/FA370610
2020-01-17 16:08:06.805 UTC [15221] LOG: restored log file "0000000100000000000000FB"…
2020-01-17 16:08:07.008 UTC [15221] LOG: restored log file "0000000100000000000000FC"…
```

Noter que PostgreSQL a commencé par chercher s'il y avait une *timeline* supérieure à celle sur laquelle il a démarré (le défaut à partir de la v12 est `recovery_target_timeline = latest`).

Ne trouvant pas ce fichier, il a commencé par rejouer ce qu'il a trouvé sur le *timeline* 1, à partir du fichier `000000010000000000000FA` (cela correspond au *checkpoint* indiqué dans `backup_label`).

- S'il y a des problème de droits, les corriger sur le répertoire `data` et les fichiers qu'il contient :

```
$ chown -R postgres /var/lib/pgsql/14/data
$ chmod -R 700 /var/lib/pgsql/14/data
```

- Le message suivant apparaît quand le fichier `backup_label` est présent et que l'on a oublié de créer `recovery.signal`. En effet, PostgreSQL ne peut pas savoir s'il a crashé au milieu d'une sauvegarde et doit redémarrer sur les journaux présents, ou s'il doit restaurer des journaux avec `restore_command`, ou encore s'il doit se considérer comme un serveur secondaire. Le message indique bien la procédure à suivre :

```
2020-01-17 16:01:16.595 UTC [14188] FATAL:  could not locate required checkpoint
                                            record
2020-01-17 16:01:16.595 UTC [14188] HINT:  If you are restoring from a backup,
        touch "/var/lib/pgsql/14/data/recovery.signal" and add required
        recovery options.
       If you are not restoring from a backup,
        try removing the file "/var/lib/pgsql/14/data/backup_label".
       Be careful: removing "/var/lib/pgsql/14/data/backup_label" will result
        in a corrupt cluster if restoring from a backup.
2020-01-17 16:01:16.597 UTC [14185] LOG:  startup process (PID 14188) exited
                                           with exit code 1
```

Il est impératif de vérifier que le point de cohérence a été atteint : il s'agit du moment où `pg_stop_backup` a marqué dans les journaux la fin du *base backup*. On pourrait arrêter le rejeu à partir d'ici au besoin.

```
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
```

Si le point de cohérence n'est pas atteint, l'instance est inutilisable ! Elle attendra indéfiniment de nouveaux journaux. C'est notamment le cas si `pg_stop_backup` a été oublié, et que PostgreSQL ne sait pas quand il doit s'arrêter !

Il est même possible de voir la présence du point de cohérence à la fin du fichier journal lui-même :

```
$ /usr/pgsql-14/bin/pg_waldump   /opt/pgsql/archives/000000010000000100000032
```

```
rmgr: Heap        len (rec/tot):     79/    79, tx:     270079, lsn: 1/32BFDC48,
                  prev 1/32BFDC00, desc: INSERT off 37 flags 0x00,
                  blkref #0: rel 1663/16384/16399 blk 1766
rmgr: Transaction len (rec/tot):     34/    34, tx:     270082, lsn: 1/32BFDC98,
                  prev 1/32BFDC48, desc: COMMIT 2020-01-17 14:58:16.102856 UTC
rmgr: Heap        len (rec/tot):     79/    79, tx:     270083, lsn: 1/32BFDCC0,
                  prev 1/32BFDC98, desc: INSERT off 41 flags 0x00,
                  blkref #0: rel 1663/16384/16399 blk 1764
rmgr: Transaction len (rec/tot):     34/    34, tx:     270079, lsn: 1/32BFDD10,
                  prev 1/32BFDCC0, desc: COMMIT 2020-01-17 14:58:16.102976 UTC
rmgr: Transaction len (rec/tot):     34/    34, tx:     270083, lsn: 1/32BFDD38,
                  prev 1/32BFDD10, desc: COMMIT 2020-01-17 14:58:16.103000 UTC
rmgr: XLOG        len (rec/tot):     34/    34, tx:          0, lsn: 1/32BFDD60,
                  prev 1/32BFDD38, desc: BACKUP_END 0/FA370610
rmgr: XLOG        len (rec/tot):     24/    24, tx:          0, lsn: 1/32BFDD88,
                  prev 1/32BFDD60, desc: SWITCH
```

Le rejeu des derniers journaux mène aux messages suivants :

```
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
```

Cela correspond à la constatation que le dernier journal est incomplet (c'est celui récupéré plus haut à la main), la recherche en vain d'une *timeline* supérieure à suivre (le fichier aurait pu apparaître entre temps), puis la création de la nouvelle *timeline*.

> Quels sont les fichiers, disparus, modifiés ?

Une fois la restauration terminée :

- `backup_label` a été archivé en `backup_label.old` ;
- à partir de la v12 : `recovery.signal` a disparu ;
- jusqu'en v11 comprise : `recovery.conf` a été archivé en `recovery.done`.

Enfin, le fichier `pg_wal/00000002.history` contient le point de divergence entre les deux *timelines* :

```
# cat 00000002.history
1       1/3F0000A0      no recovery target specified
```

Le `postgresql.auto.conf` peut conserver la `recovery_command` si l'on est sûr que ce sera la bonne à la prochaine restauration.

> Vérifier que la connexion peut se faire.
> Générer des journaux : quel est leur nom ?

```
$ psql pgbench
psql (14.1)
Saisissez « help » pour l'aide.

pgbench=# VACUUM ;
```

Les journaux générés appartiennent bien à la *timeline* 2 :

```
$ ls -altr data/pg_wal/

total 737244
-rw-------.  1 postgres postgres 16777216 17 janv. 16:08 000000010000000010000003F
drwx------. 20 postgres postgres     4096 17 janv. 16:08 ..
-rw-------.  1 postgres postgres       42 17 janv. 16:08 00000002.history
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 000000020000000010000003F
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000040
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000041
…
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000064
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000065
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000066
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000067
-rw-------.  1 postgres postgres 16777216 17 janv. 16:40 0000000200000000100000068
```

> Récupérer la dernière valeur de `pgbench_history.mtime`. Vérifier qu'elle est supérieure à celle relevée à la fin de la sauvegarde.

```
SELECT max(mtime) FROM pgbench_history ;

            max
----------------------------
 2020-01-17 14:59:30.092409
```

1.11.3 PG_BASEBACKUP : SAUVEGARDE PONCTUELLE & RESTAURATION

> Configurer la réplication dans `postgresql.conf` : activer l'archivage, autoriser des connexions en streaming.

Si l'archivage est actif, ici on choisit de ne pas archiver réellement et de ne passer que par la réplication :

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

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                                     trust
```

Redémarrer PostgreSQL :

```
# systemctl restart postgresql-14
```

> Insérer des données et générer de l'activité avec `pgbench`.
> En parallèle, sauvegarder l'instance avec `pg_basebackup` au format tar, sans oublier les journaux, et avec l'option `--max-rate=16M` pour ralentir la sauvegarde.

```
$ createdb bench
$ /usr/pgsql-14/bin/pgbench -i -s 100 bench
$ vacuumdb -az
$ psql -c "CHECKPOINT;"
$ /usr/pgsql-14/bin/pgbench bench -n -P 5 -T 360

$ pg_basebackup -D /var/lib/pgsql/14/backups/basebackup -Ft \
--checkpoint=fast --gzip --progress --max-rate=16M

1567192/1567192 kB (100%), 1/1 tablespace
```

> Tout au long de l'exécution de la sauvegarde, surveiller l'évolution de l'activité sur la table `pgbench_history`.

```
$ watch -n 5 "psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'"
```

> Une fois la sauvegarde terminée, vérifier son état ainsi que la dernière donnée modifiée dans `pgbench_history`.

```
$ ls -lha /var/lib/pgsql/14/backups/basebackup
(...)
-rw-------. 1 postgres postgres 86M Nov 29 14:25 base.tar.gz
-rw-------. 1 postgres postgres 32M Nov 29 14:25 pg_wal.tar.gz

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

            max
----------------------------
 2019-11-29 14:27:21.673308
```

> Arrêter l'instance.
> Par sécurité, faites une copie à froid des données (par exemple avec `cp -rfp`).
> Vider le répertoire.
> Restaurer la sauvegarde `pg_basebackup` en décompressant ses deux archives.

```
# systemctl stop postgresql-14
# cp -rfp /var/lib/pgsql/14/data /var/lib/pgsql/14/data.old
# rm -rf /var/lib/pgsql/14/data/*
# tar -C /var/lib/pgsql/14/data \
      -xzf /var/lib/pgsql/14/backups/basebackup/base.tar.gz
# tar -C /var/lib/pgsql/14/data/pg_wal \
      -xzf /var/lib/pgsql/14/backups/basebackup/pg_wal.tar.gz
# rm -rf /var/lib/pgsql/14/data/log/*
# systemctl start postgresql-14
```

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

```
# tail -f /var/lib/pgsql/14/data/log/postgresql-*.log
2019-11-29 14:29:47.733 CET [7732] LOG: database system was interrupted;
last known up at 2019-11-29 14:24:10 CET
2019-11-29 14:29:47.811 CET [7732] LOG: redo starts at 4/3B000028
2019-11-29 14:29:48.489 CET [7732] LOG:
                                   consistent recovery state reached at 4/4A960188
2019-11-29 14:29:48.489 CET [7732] LOG: redo done at 4/4A960188
2019-11-29 14:29:55.105 CET [7729] LOG:
                                   database system is ready to accept connections
```

> Quelle est la dernière donnée restaurée ?

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

```
            max
----------------------------
 2019-11-29 14:25:45.698596
```

Grâce aux journaux (`pg_wal`) restaurés, l'ensemble des modifications survenues lors de la sauvegarde ont bien été récupérées.

Toutefois, 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.

```
# systemctl stop postgresql-14
# rm -rf /var/lib/pgsql/14/data/*
# tar -C /var/lib/pgsql/14/data \
      -xzf /var/lib/pgsql/14/backups/basebackup/base.tar.gz
# rm -rf /var/lib/pgsql/14/data/log/*
# systemctl start postgresql-14

# cat /var/lib/pgsql/14/data/log/postgresql-*.log
2019-11-29 14:44:14.958 CET [7856] LOG:
           database system was shut down in recovery at 2019-11-29 14:43:59 CET
2019-11-29 14:44:14.958 CET [7856] LOG:  invalid checkpoint record
2019-11-29 14:44:14.958 CET [7856]
           FATAL:  could not locate required checkpoint record
2019-11-29 14:44:14.958 CET [7856] HINT:  If you are restoring from a backup,
    touch "/var/lib/pgsql/14/data/recovery.signal" and add required recovery options.
    If you are not restoring from a backup, try removing the file
    "/var/lib/pgsql/14/data/backup_label".
    Be careful: removing "/var/lib/pgsql/14/data/backup_label" will result
    in a corrupt cluster if restoring from a backup.
2019-11-29 14:44:14.959 CET [7853] LOG: startup process (PID 7856) exited
                                       with exit code 1
2019-11-29 14:44:14.959 CET [7853] LOG: aborting startup due to startup
                                       process failure
2019-11-29 14:44:14.960 CET [7853] LOG: database system is shut down
```

PostgreSQL ne trouvant pas les journaux nécessaires à sa restauration à un état cohérent, le service refuse de démarrer. Supprimer le fichier `backup_label` permettrait toutefois de démarrer l'instance MAIS celle-ci serait alors dans un état incohérent.

1.11.4 PG_BASEBACKUP : SAUVEGARDE PONCTUELLE & RESTAURATION

> Remettre en place la copie de l'instance prise précédemment.
> Configurer l'archivage.

```
# systemctl stop postgresql-14
# rm -rf /var/lib/pgsql/14/data
# cp -rfp /var/lib/pgsql/14/data.old /var/lib/pgsql/14/data
```

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

```
$ mkdir /var/lib/pgsql/14/archives
```

L'utilisateur système `postgres` doit avoir le droit d'y écrire.

Ensuite, dans `postgresql.conf` :

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

# systemctl start postgresql-14
# rm -rf /var/lib/pgsql/14/backups/basebackup/*
```

> Générer à nouveau de l'activité avec `pgbench`.
> En parallèle, lancer une nouvelle sauvegarde avec `pg_basebackup`.

```
$ psql -c "CHECKPOINT;"
$ /usr/pgsql-14/bin/pgbench bench -n -P 5 -T 360
```

En parallèle, lancer à nouveau `pg_basebackup` :

```
$ pg_basebackup -D /var/lib/pgsql/14/backups/basebackup -Ft \
--checkpoint=fast --gzip --progress --max-rate=16M
1567192/1567192 kB (100%), 1/1 tablespace
```

> Vérifier que des archives sont bien générées.

```
$ ls -lha /var/lib/pgsql/14/archives
(...)
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005C
-rw-------. 1 postgres postgres  340 Nov 29 14:55
                                        000000010000000040000005C.00002088.backup
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005D
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005E
-rw-------. 1 postgres postgres  16M Nov 29 14:54 000000010000000040000005F
-rw-------. 1 postgres postgres  16M Nov 29 14:54 0000000100000000400000060
-rw-------. 1 postgres postgres  16M Nov 29 14:54 0000000100000000400000061
(...)
```

> Effacer le PGDATA.
> Restaurer la sauvegarde, configurer la `restore_command` et créer le fichier `recovery.signal`.

```
# systemctl stop postgresql-14
# rm -rf /var/lib/pgsql/14/data/*
# tar -C /var/lib/pgsql/14/data \
  -xzf /var/lib/pgsql/14/backups/basebackup/base.tar.gz
# rm -rf /var/lib/pgsql/14/data/log/*
```

Configurer la `restore_command` dans le fichier `postgresql.conf` :

```
restore_command = 'rsync /var/lib/pgsql/14/archives/%f %p'
```

Créer le fichier `recovery.signal` :

```
$ touch /var/lib/pgsql/14/data/recovery.signal
```

Démarrer le service :

```
# systemctl start postgresql-14
```

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

```
# tail -f /var/lib/pgsql/14/data/log/postgresql-*.log
2019-11-29 15:02:34.736 CET [8280] LOG:  database system was interrupted;
last known up at 2019-11-29 14:54:19 CET
2019-11-29 15:02:34.808 CET [8280] LOG: starting archive recovery
2019-11-29 15:02:34.903 CET [8280] LOG: restored log file "0000000100000040000005C"
                                       from archive
2019-11-29 15:02:34.953 CET [8280] LOG: redo starts at 4/5C002088
2019-11-29 15:02:35.083 CET [8280] LOG: restored log file "0000000100000040000005D"…
(...)
2019-11-29 15:02:37.254 CET [8280] LOG: restored log file "00000001000000400000069"…
2019-11-29 15:02:37.328 CET [8280] LOG:
                                     consistent recovery state reached at 4/69AD0728
2019-11-29 15:02:37.328 CET [8277] LOG: database system is ready to accept
                                       read only connections
2019-11-29 15:02:37.430 CET [8280] LOG: restored log file "0000000100000040000006A"…
2019-11-29 15:02:37.614 CET [8280] LOG: restored log file "0000000100000040000006B"…
(...)
2019-11-29 15:03:08.599 CET [8280] LOG: restored log file "00000001000000400000096"…
2019-11-29 15:03:08.810 CET [8280] LOG: redo done at 4/9694CA30
2019-11-29 15:03:08.810 CET [8280] LOG:
           last completed transaction was at log time 2019-11-29 15:00:06.315365+01
```

```
2019-11-29 15:03:09.087 CET [8280] LOG: selected new timeline ID: 2
2019-11-29 15:03:09.242 CET [8280] LOG: archive recovery complete
2019-11-29 15:03:15.571 CET [8277] LOG: database system is ready
                                        to accept connections
```

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

```
$ psql -d bench -c 'SELECT max(mtime) FROM pgbench_history;'
            max
----------------------------
 2019-11-29 15:00:06.313512
```

NOTES

NOTES

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **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

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.